Thesis for the Degree of Licentiate of Philosophy

Pure Functional Parsing
an advanced tutorial

Peter Ljunglöf

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, March 2002

A dissertation for the Licentiate Degree in Computing Science at
Göteborg University and Chalmers University of Technology

Technical Report no. 6L
School of Computer Science and Engineering

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2002

Abstract

Parsing is the problem of deciding whether a sequence of tokens is recognized by a given grammar, and in that case returning the grammatical structure of the sequence.

This thesis investigates different aspects of the parsing problem from the viewpoint of a functional programmer. It is conceptually divided into two parts, discussing the parsing problem from different perspectives; first as a comprehensive survey of possible implementations of combinator parsers; and second as pure functional implementations of standard context-free parsing algorithms.

The first part of the thesis is a survey of the possible implementations of combinator parsers that have previously been suggested in the litterature, relating their dirrefent usages. A number of previously unknown parser implementations are also given, especially efficient for small and medium-sized natural language applications.

The second part of the thesis define elegant and declarative, pure functional versions of some standard parsing algorithms for context-free grammars. The goal has been to implement the algorithms in a way that is close to their intuitive formulations, not sacrificing computational efficiency. The implementations only use simple data structures not relying on a global updateable state, thus opening the way for nice functional implementations.

Finally the thesis implements parser combinators that can collect the grammatical structure in the program, to be able to use any suitable parsing algorithm and not just recursive descent. However, this requires an mildly impure extension of the host language Haskell.

Keywords: *context-free grammars, parsing algorithms, parser combinators, functional programming.*

Acknowledgements

Although I'm the sole writer of the text in this thesis, it would never have been written without the support of a huge number of people.

First of all I would like to thank everyone who has helped me directly with the making of the thesis. My supervisor Aarne Ranta has always believed in me, even though my attention is constantly diverted by other interesting subjects. But I also owe many thanks to all people who have been the target of my stupid questions, who have patiently explained the most basic concepts, who have dissected my silly ideas, and who have gladly entered into discussions on any concievable subject; especially Koen Claessen, Niklas Eén, Jörgen Gustavsson, John Hughes, Patrik Jansson, Josef Svenningsson, Tuomo Takkula and Dag Wedelin. I am also grateful to have Paul Callaghan as my opponent, with many insightful comments on preliminary versions, and for the patience with all my exceeded deadlines.

Secondly I would like to thank the people who have helped me through these years more indirect; Robin Cooper and Jan Smith for making me interested in doing research in the first place; the members of the Language Technology group here at the department, and our collaborators at Linguistics and Philosophy, for all the seminars; my office mate Karol Ostrovsky and the other fellow PhD students for all the social events I almost certainly fail to attend; and the rest of the people at this department, both researchers and administrative.

Thirdly I would like to thank the outside world; all my friends who still fail to understand what I do for a living; and my beloved family Saga, Signe and Svea, who always remind me that there really is a world outside of this department. Without you I would have been a finished PhD several years ago. . . I love you beyond infinity.

Contents

Chapter 1

Introduction

Parsing is one of the fundamental areas of computation. It is necessary both for programming languages and natural languages. Therefore it is an area for which much research has been conducted, and efficient parsing algorithms for various kinds of languages has been developed.

It is well known that parsing can be performed in time linear in the size of the input for certain sub-classes of context-free grammars. For general context-free grammars it is possible to parse in cubic time.

Unfortunately, for us functional programmers, there has not been much research conducted from a functional viewpoint. Almost all algorithms are described in an imperative manner, and it is not always easy to transform an imperative algorithm into a functional style, especially when using a pure language such as Haskell.

Most parsing research for functional languages has been on parser combinators, which is a nice and clean way of writing context-free grammars (and grammars in more general grammatical formalisms), automatically yielding a parser for that grammar. Unfortunately, combinator parsers can only implement one specific parsing algorithm called recursive descent. And this particular algorithm is not especially well suited for general context-free grammars. On some sub-classes of grammars it can be quite efficient though.

1.1 About the thesis

This thesis is all about parsing in pure functional programming languages. One can say that the thesis is divided into two different parts, which are tied together in chapter 8.

The thesis is laid out as an advanced tutorial on the concept of parsing for functional programmers. This means that there are some exercises for the reader throughout the thesis.

Part I: Parser combinators The first part – chapters 3 and 4 (and the second half of chapter 2) – deals with combinator parsers. One can say that it is a survey of different possible implementations of the type of combinator parsers. Chapter 2 defines the basic parser combinators and discusses some relations between the context-free combinators and the monadic ones. In chapter 3 we give the standard types of depth-first searching parsers, with and without continuation transformers. We also introduce the breadth-first searching stream processor parsers, which come with or without continuation transformers. Chapter 4 introduces the data structure of tries, or letter trees, which becomes a left-factorizing kind of parser. The chapter explores different types of tries for more efficient sharing of sub-tries, thus turning the regular parts of the letter trees into finite automata. The chapter ends by describing a trie that instead of returning a normal parse result returns a special-purpose standard parser.

Part II: Parsing algorithms The second part – chapters 5, 6 and 7 (and the first half of chapter 2) – deals with general context-free parsing algorithms. To be able to implement a parsing algorithm which is not recursive descent, one needs to have the grammar as a specific object which can be passed around and examined by the parsing algorithm. Chapter 2 defines what a context-free grammar is, together with a suitable Haskell implementation. Chapter 5 implements an efficient functional version of bottom-up chart parsing, which builds a directed graph where the edges are labelled with syntactic categories. Chapter 6 implements an approximation of the general Tomita LR parsing algorithm, where the grammar is pre-compiled into an LR table to speed up the parsing. The approximation stems from the fact that we have simplified the original complicated data structures. Chapter 7 describes an implementation of CYK parsing, where we see a context-free grammar as a multiplicative operator and then implement parsing as calculating the transitive closure of a matrix consisting of syntactic categories.

Tying the parts together Finally, in chapter 8 we use an extension of Haskell to implement parser combinators that can collect the grammatical structure in the program. Then we can use any good old parsing algorithm to parse the grammar, and transform the parse trees into the expected parse results.

The resulting combinator parser is compared with the alternative approach of using a parser generator such as Happy, describing advantages as well as disadvantages.

Final discussion The last chapter, chapter 9, is a final summary of the contents of the thesis, together with a short discussion on some implications. We compare the two approaches in the thesis – combinator parsers and parsing algorithms – and relate them to their main competitor, parser generators. The chapter also consits of a summary of test results. The different combinator parsers and parsing algorithms have been tested on different kinds of data to see which parser is suitable for which kind of input.

There are also two appendices. Appendix A consists of an example implementation of sets and finite maps, which are abstract data types that we use heavily throughout the thesis. Appendix B consists of all the test results that are summarized in the final discussion.

1.2 Prerequisites

To have benefit of the thesis the reader is supposed to know some things beforehand.

Functional programming The reader is supposed to be a skilled functional programmer.

Laziness Since laziness is used in many places, the reader should at least have some familiarity with the concept of lazy functional programming.

Types and type classes We will use the Haskell type system, together with some extensions. This means that the reader should know something about the standard Hindley-Milner type system, and it will help much to know the ideas of the type classes used in Haskell for overloading functions.

Standard Haskell functions We use some standard Haskell functions without mentioning, such as *foldr*, *map*, function composition $(\cdot)$, et cetera.

List comprehensions and pattern guards We will use list comprehension syntax which is a standard part of Haskell. Also, guards in pattern matching is a handy syntactic sugar.

1.2.1 Notational conventions

We use the pure functional language Haskell throughout the thesis. For people not so familiar with its syntax, we here give some small hints. For more extensive information we refer to the Haskell 98 Language Report [35] or to the Gentle Introduction [15].

Types and type constructors start with capital letters while constants and variables start with lower-case letters. Functions written with non-alphanumeric symbols are automatically infix operators. An ordinary function can be converted to an operator by surrounding it with backticks (e.g. $3\,`elem`\,[1\,..\,5] \equiv elem\ 3\ [1\,..\,5]$), and operators surrounded by parentheses become standard functions (e.g. $(+)\ 3\ 4 \equiv 3 + 4$). The operator precedence and associativity can be declared by **infix**, **infixr** and **infixl**, where the precedence is a number between 0 and 9 (higher binds harder).

Haskell makes heavy use of layout – instead of surrounding a block in braces, one usually just use indentation instead. Local declarations can be made with standard **let** declarations or, more commonly, via a **where** declaration in the end of the definition.

Types can be declared in three ways; **type** $\tau = \alpha$ declares a type synonym, and **data** $\tau = \alpha$ declares a new algebraic datatype. If the data type has one single constructor with only one argument, **newtype** can be used instead of **data** to make it possible for the compiler to generate more efficient code.

In this thesis constants and variables are written in italics (*function*), types and type constructors in a sans-serif font (Type), and reserved words are written in a bold font (**class**).

1.2.2 Extensions to Haskell

Existential quantification in types Existential quantification in data types is an extension to standard Haskell, but which is recognized by all existing Haskell compilers and interpreters. The meaning of the data type declaration **data** T = C $(\forall\alpha.\ \tau)$ is that the C constructor can take an object of type τ for any possible type α. The construction $\forall x.\ \tau$ which is used in this thesis is written "`forall x .` τ" in the Haskell extensions.

The extension is simple and well-behaved, and is described in e.g. [9, 25, 33]. The explanation for the name "existential" quantification is that in the type of the constructor C the argument α is existentially quantified over. Or in other words, $(\forall\alpha.\tau) \to T \equiv \exists\alpha.(\tau \to T)$.

Multi-parameter type classes with functional dependencies Multi-parameter type classes are a natural generalization of the Haskell class system. Unfortunately it is not very useful as it stands, since many of the interesting instance declarations will overlap, which is not allowed in Haskell. Functional dependencies, introduced by Jones [18], is a way to overcome this limitation. This means that we can say that one of the parameters depends entirely on one of the other parameters.

class Parser $m\ s\ \mid\ m \to s$ **where**...

The above definition says that both the parser type m and the type of input symbols s are parameters in this class definition, but the symbol type must depend on the parser type. This means that the parser must refer to the type of symbols somewhere in its type, but it is not interesting to know exactly where.

1.3 Contributions of the thesis

This thesis looks at some issues in the area of parsing for functional programming languages. To our knowledge it is the first extensive study relating different types of combinator parsers. It also defines elegant and purely functional versions of some standard parsing algorithms. Finally it relates the two concepts by defining a combinator parser that can collect the grammatical structure and then call any suitable parsing algorithm.

The chapters describe real implementations of parsers, not only abstract algorithms. The implementations are collected in a library of combinator parsers and parsing algorithms, which can be found in the Chalmers Multi Library on the World Wide Web:

`http://www.cs.chalmers.se/Cs/Research/Functional/MultiLib`

1.3.1 Part I: Parser combinators

There has until now not been any work that tries to relate the different combinator parsers suggested previously. In this part we do exactly this – relate different kinds of parsers to each other, and define general parser transformers that apply to several parsers. We also define some new kinds of parsers – the trie based combinator parsers.

Chapter 2: Grammars and parsers We define a class hierarchy for working with the two different kinds of combinator parsers that exist – context-free and monadic parsers. The monadic combinators are more expressive than the context-free, but some parsers are impossible to make monadic. We also show some trivial results relating the context-free combinators with context-free grammars.

Chapter 3: Existing parser combinators We define the standard depth-first searching parsers, which are descended from Wadler's original parser type [44]. On these we can apply either the continuation transformer or the stack continuation transformer. The latter is derived from Swierstra's efficient error-correcting parser [39, 40].

Another class of parsers is the breadth-first searching stream processor parser, originally coming from the graphics library Fudgets [3]. That the stream processor also is a parser has been noticed by Claessen [5]. The continuation transformers can also be applied to the stream parser.

Chapter 4: Left-factorizing parser combinators From the data structure of tries, or letter trees, we define several different parsers. These parsers are all left-factorizing, which means that it is not necessary to do the left-factorization by hand. We examine the memory behaviour and extend the original parser getting a parser with better memory behaviour.

Finally it is possible to combine the left-factorization of the trie structure with the efficiency of the standard depth-first parsers. This was originally done by Swierstra in his efficient parser [39, 41], but now we have split that complicated parser type into a couple of less complicated parser transformers.

1.3.2 Part II: Parsing algorithms

The second part of the thesis shows how to implement some of the main general context-free parsing algorithms in a pure functional style. This means that it is not necessary to pass around a global mutable parsing state, such as a chart (in chart parsing), a graph-structured stack (in LR parsing) or a parse matrix (in CYK parsing). Instead we show how to implement these data structures in a more incremental way than traditionally, thus yielding pure, simple and elegant functional implementations.

Chapter 2: Grammars and parsers We define the basic notions for context-free grammars and show how to implement these notions in a functional language.

Chapter 5: Chart parsing Chart parsing is really a family of parsing algorithms, all based on the data structure of charts, which is a set of known facts called edges. The parsing process uses inference rules to add new edges to the chart, and the parsing is complete when no further edges can be added.

We implement an efficient functional version of bottom-up Kilbury chart parsing [20, 46]. The novel thing is that it doesn't have to rely on a global state for the implementation of the chart. This makes the code clean, elegant and declarative, while still being as efficient as the standard imperative implementations. The worst-case complexity is cubic in the length of the input, which is as good as one can get in a practical implementation.

Chapter 6: Generalized LR parsing LR parsing is a way of pre-compiling the grammar into parse tables, which then can be used while parsing. Originally LR parsing was only used for deterministic grammars, but Lang and Tomita showed how to use the tables to efficiently parse general context-free grammars [24, 42]. This is often called generalized LR parsing.

We describe several different versions of generalized LR parsing. The most interesting is the Tomita-like parser, where the originally complicated graph-structured stack has been simplified to a tree structure, so that we do not have to implement it using a mutable global state. The graph structure of the stack will still be stored in Haskell memory, because equivalent sub-trees will be shared.

A major drawback is that the algorithm will not be polynomial in time, which the original Tomita algorithm is. The reason is that it is impossible from inside Haskell to know that two sub-trees are shared or not, and then the algorithm will do double work while traversing the stack.

Chapter 7: CYK parsing The parsing strategy of Cocke, Younger and Kasami [19, 47] is often described as matrix multiplication, but the actual implementations seldom bears any resemblance with the higher-level descriptions. When using a high-level language such as Haskell, we can retain the idiom. In fact this leads to an elegant and purely functional version of CYK parsing, which is as efficient as the standard imperative versions. The worst-case complexity is cubic in the length of the input, which is as good as one can hope to get.

1.3.3 Tying the parts together

There are some advantages with parser combinators (in part I) which is lacking when we implement a grammar as a Haskell object (in part II). One important feature is that we can define new combinators that makes it easier to implement grammars. Examples include variants of the *foldr* and *foldl* functions, combined with *intersperse*. Another feature is that we can define semantic actions for the productions directly, instead of going via an intermediate result type. A third feature is that the Haskell type checker can catch errors in the grammar, which can be very difficult to find with an untyped grammar. A final feature is that many parser types are monadic, which makes it possible to parse non context-free languages.

The main disadvantage with parser combinators is that they all implement the same parsing algorithm – recursive-descent parsing, which is not the most efficient parsing algorithm. This algorithm is also not capable of parsing all possible grammars – left-recursive grammars will lead to non-termination, which is a seriuos problem since left-recursion is very common in both formal and natural languages.

One partial solution to this problem is to use a parser generator, such as Happy [28]. With a parser generator we can write a context-free grammar together with Haskell code that produces semantic actions of the productions. The grammar (or rather, the semantic actions) is also type-checked, which the grammar-as-Haskell-object approach is not. The grammar is converted by Happy to an efficient parser implemented in Haskell.

Unfortunately there are two problems with parser generators. First, we still lack the first feature of parser combinators mentioned above – we cannot define our own new combinators to simplify grammar writing. Second, we need to learn a new language – the Happy language, which is an extension of Haskell (although a small one).

Chapter 8: Grammar-collecting parser combinators Until now there has not been any way of transforming a grammar defined by a combinator parser to a context-free grammar. In standard Haskell this is not possible, since we cannot stop ourselves from falling into an infinite loop while examining the grammar structure.[1]

In this chapter we show one possible way of accomplishing this, by using the mildly impure extension of "observable sharing", developed by Claessen and Sands [6]. We define a combinator parser that collects the structure of the grammar. This means that we can use any good parsing algorithm, not just recursive descent, in a combinator parser; as well as checking properties of the grammar, such as if it is LL or LR. The parser is not monadic, which means that the fourth feature of combinator parsers do not apply.

With this extension we can say that we have extended the idea of parser combinators to a more general idea of "grammar combinators".

But there are disadvantages too, of which the most important is that we must be very careful when defining new combinators. We have to make sure that the resulting grammar is finite, and that we do not lose vital sharing information; otherwise the collection procedure might never terminate.

1.4 How to read the thesis

Before reading the other chapters, it is necessary to read chapter 2, which defines the notions, types and type classes that are used throughout the thesis.

The rest of the chapters are pretty much independent of each other, with some exceptions. Both chapter 4 and in some extent chapter 8 depend on the contents of chapter 3; and to benefit from chapter 9 it is good to have at least skimmed through the previous chapters.

[1]At least it is not possible while staying inside Haskell – one can always parse the grammar file with a parser for Haskell, and extract the grammar. But then we have to limit the language of the grammar to a subset of Haskell, and lose the first of the features of parser combinators.

$$\begin{tikzcd}
 & & 3 \arrow[r] \arrow[dr] & 4 \arrow[rrdd, dotted] & & \\
 & & & 8 \arrow[rrd, dotted] & & \\
1 \arrow[r] & 2 \arrow[r] \arrow[uur] \arrow[dr] \arrow[ddr] & 5 \arrow[rrr, dotted] & & & 9 \\
 & & 6 \arrow[rrru, dotted] & & & \\
 & & 7 \arrow[rrruu, dotted] & & &
\end{tikzcd}$$

Chapter 2

Grammars and Parsers

This chapter is an introduction to context-free grammars and parser combinators. We also show some correspondences between the two concepts. We describe two kinds of combinator parsers – context-free and monadic, the latter being more general than the former.

2.1 Context-free grammars

The standard way to define a context-free grammar is as a tuple $G = \langle N, \Sigma, P, S \rangle$, where N and Σ are disjoint sets of *nonterminal* and *terminal* symbols respectively, P is a set of *productions* and $S \in N$ is the *start symbol*. The nonterminals are also called *categories* and the set $V = N \cup \Sigma$ are the *symbols* of the grammar. Each production in P is a pair $\langle A, \alpha \rangle$, where $A \in N$ is a nonterminal and $\alpha \in V^*$ is a sequence of symbols. The sets N, Σ and P are all finite.

We use capital letters A, B, C for nonterminals, lower-case letters s, t, u for terminal symbols, and lower-case a, b, c for general symbols (elements in V). Greek letters α, β, γ will be used for sequences of symbols, and we write ϵ for the empty sequence. We usually write $A \to \alpha$ if $\langle A, \alpha \rangle \in P$; a production $A \to \epsilon$ is called an empty production, and $A \to b$ is called a unit production.

The relation $\Rightarrow$ is defined by $\alpha B \gamma \Rightarrow \alpha \beta \gamma$ iff $B \to \beta$. We will not make use of this relation directly, but instead the transitive closure $\Rightarrow^+$ and the reflexive and transitive closure $\Rightarrow^*$ (defined as usual by $\alpha \Rightarrow^+ \beta$ iff $\alpha \Rightarrow \cdots \Rightarrow \beta$; and $\alpha \Rightarrow^* \beta$ iff $\alpha = \beta$ or $\alpha \Rightarrow^+ \beta$).

A *phrase* is a sequence of terminals $\beta \in \Sigma^*$ such that $A \Rightarrow^+ \beta$ for some $A \in N$; sometimes we will call this an A phrase. A *sentence* is a phrase recognized by the start symbol, i.e. an S phrase. The *language* accepted by a grammar is the set of sentences of that grammar. A grammar is *left-recursive* if $A \Rightarrow^+ A\alpha$ for some $A \in N$, and furthermore *hidden* left-recursive if $A \Rightarrow^+ BA\alpha$, where

$B \Rightarrow^+ \epsilon$. The set of *empty categories* is $\{ A \in N \mid A \Rightarrow^* \epsilon \}$, the *left corners* of a sequence of symbols α are $\{ b \in V \mid \alpha \Rightarrow^* b\beta \}$, and the *follow* set of a symbol b is $\{ c \in V \mid \alpha \Rightarrow^* \beta bc\gamma \}$.[1]

The grammar is assumed to be cycle-free (which means that there is no A such that $A \Rightarrow^+ A$), which is a standard assumption since cycles are not productive and lead to infinitely many ways to recognize a sentence. We will also assume that the terminal symbols only appear in unit productions. This will constitute no severe drawback, since any grammar can be easily transformed into this form by adding a new category and a unit production for each terminal, changing the rest of the productions slightly.

One particular subclass of context-free grammars, which will be used in chapter 7, is Chomsky normal form. A grammar is in Chomsky normal form when all productions are either on the form $A \to t$ or on the form $A \to BC$, where t is a terminal and A, B and C are nonterminals. It is always possible to transform a context-free grammar into Chomsky normal form, and still accept the same language.[2] If the original grammar has n productions, the transformed version can have in the worst case $O(n^2)$ productions [13].

Another possible language-preserving transformation is to remove left-recursion. Left-recursive grammars lead to non-termination of some parsing algorithms, most notably recursive descent parsing which is the parsing algorithm implemented by parser combinators. We do not describe how the transformations are done here, but refer to any standard textbook about formal languages, such as [1, 13].

2.2 Sets and finite maps in Haskell

Thoroughout the thesis we will make use of sets and finite maps. These types occur in both combinator parsers and parsing algorithms, even in the definition of a grammar. In our Haskell implementations we leave sets and finite maps as abstract data types, and give example implementations in appendix A. The example implementation of sets are as ordered lists, and the finite maps are given as ordered association lists.

[1]The set of left corners can also be called the *first set*, with the difference that the first set is usually only defined for terminal symbols. In the same way we deviate slightly from the traditional definition of the follow set – our definition can have any kind of symbol in the set.

[2]With the exception of grammars that accept ϵ, which grammars in Chomsky normal form cannot accept. But this is not a real limitation.

2.2.1 Sets

The type of sets is abstract; and sets can be compared for equality.

```
data Set α = ...
instance Ord α ⇒ Eq (Set α) where
  (==) = ...
```

Most implementations of sets are only possible if the element type have an ordering, so we use the context Ord α on each of the set operations. Observe that most of these operations can be defined in terms of other operations, but we leave these definitions as exercises for the reader.

There is an empty set and we can create the singleton set from a single element.

```
emptySet  :: Ord α ⇒ Set α
unitSet   :: Ord α ⇒ α → Set α
```

We have a test for emptiness and for membership in a set.

```
isEmptySet :: Ord α ⇒ Set α → Bool
elemSet    :: Ord α ⇒ α → Set α → Bool
```

We can join two sets into one, and form the union of a list of sets.

```
(<++>) :: Ord α ⇒ Set α → Set α → Set α
union  :: Ord α ⇒ [Set α] → Set α
```

We can create a set from a list of elements, and extract the list of elements from a set.

```
makeSet :: Ord α ⇒ [α] → Set α
elems   :: Ord α ⇒ Set α → [α]
```

We will assume that *elems* returns an ordered list. In some cases we want to create a set from a list which we know is ordered and without duplicates; many implementations can do this in a faster way, and therefore we add an extra function to create a set from an already ordered list.

```
ordSet :: Ord α ⇒ [α] → Set α
```

Finally we often want to build a set from an initial set and a function that yields new elements from already added elements. The *limit* function builds the minimal fix-point set.

```
limit :: Ord α ⇒ (α → Set α) → Set α → Set α
```

If we assume that we have an operation (<—>) for set difference, it is possible to define the *limit* function in the following way.

```
limit more start                 =  limit′ start start
  where limit′ old new
                  | isEmpty new′  =  old
                  | otherwise     =  limit′ (new′ <++> old) (new′ <—> old)
              where new′          =  union (map more new)
```

Observe that the *limit* function does not necessarily terminate – it depends on the given function *more*. But if there exists a finite fix-point set, *limit* will find the minimal one.

2.2.2 Finite maps

The type of finite maps from keys s to values α is also abstract. For the same reason as above we assume an ordering on the type of keys.

type Map $s\ \alpha = \ldots$

There is an empty map, and we can map a key to a value.

```
emptyMap  ::  Ord s ⇒ Map s α
(|→)      ::  Ord s ⇒ s → α → Map s α
```

We can test if a finite map is empty, and we can use it to lookup values. The operation (?) corresponds to the standard *lookup* function on association lists, with the arguments swapped.

```
isEmptyMap  ::  Ord s ⇒ Map s α → Bool
(?)         ::  Ord s ⇒ Map s α → s → Maybe α
```

To merge two finite maps we need to know what to do with conflicting values. Therefore *mergeWith* takes a function that combines two values into one.

```
mergeWith  ::  Ord s ⇒ (α → α → α) → Map s α → Map s α → Map s α
```

We can build a finite map from an association list, which can have duplicate keys, in which case we need to know what to do with duplicate values. We can also convert a map to an association list.

```
makeMapWith  ::  Ord s ⇒ (α → α → α) → [(s, α)] → Map s α
assocs       ::  Ord s ⇒ Map s α → [(s, α)]
```

The result of *assocs* will be ordered on the key. Most implementations can create the map faster if the given association list is already ordered on the keys, and that the keys are not duplicated.

```
ordMap  ::  Ord s ⇒ [(s, α)] → Map s α
```

Finally it is possible to map a function on the values in a finite map.[3]

$$mapMap :: \mathsf{Ord}\ s \Rightarrow (\alpha \rightarrow \beta) \rightarrow \mathsf{Map}\ s\ \alpha \rightarrow \mathsf{Map}\ s\ \beta$$

2.3 Context-free grammars in Haskell

In this section we translate the definitions in section 2.1 into Haskell code. The categories and terminals will be Haskell types; but since they depend on the grammar we abstract over these types.

The grammar Since the terminals only appear in unit productions, and also only once, we can split the set P of productions into a set of nonterminal productions and a function mapping each terminal to a set of nonterminals. The type of a context-free grammar will therefore be a four-tuple with a set of categories, the starting category, the terminal mapping and the set of productions.[4]

$$\textbf{type}\ \mathsf{Grammar}\ c\ t \quad = \quad (\mathsf{Set}\ c,\ c,\ t \rightarrow \mathsf{Set}\ c,\ \mathsf{Set}\ (\mathsf{Production}\ c))$$

Observe that we do not include the set of terminals, since we will have no use of that set in any of the algorithms presented. A production is simply a pair of a category and a list of categories.

$$\textbf{type}\ \mathsf{Production}\ c \quad = \quad (c,\ [\,c\,])$$

Parsing A parsing algorithm will be a function that, given a grammar, takes a sequence of terminals and returns a parse result. A parse result can be anything between a boolean value (in which case the parser is called a *recognizer*) and a set of semantic functions. In this thesis we will consider lists of parse trees as the result from the parsing algorithm, and an implementation of an algorithm will be implemented as the functions *recognize* or *parse*.

$$\begin{array}{lll} recognize & :: & \mathsf{Grammar}\ c\ t \rightarrow [\,t\,] \rightarrow \mathsf{Bool} \\ parse & :: & \mathsf{Grammar}\ c\ t \rightarrow [\,t\,] \rightarrow [\,\mathsf{ParseTree}\ c\ t\,] \end{array}$$

Parse trees A parse tree is either a node with a category and a sequence of parse trees, or a leaf consisting of a terminal.

$$\begin{array}{lll} \textbf{data}\ \mathsf{ParseTree}\ c\ t & = & c :\hat{}\ [\,\mathsf{ParseTree}\ c\ t\,] \\ & | & \mathsf{Leaf}\ t \end{array}$$

[3] We could also have chosen to make maps an instance of the Functor class, but we use a specific function to improve readability later.

[4] Throughout this thesis we will use the type variables c and t for the types of categories and terminals respectively.

Exercise

Write a function that translates a general context-free grammar into Chomsky normal form. The translation consists of three parts.

- Removing the empty productions. To remove an empty production $A \to \epsilon$ you must add an extra production $B \to \alpha\beta$ for each $B \to \alpha A \beta$ already in the grammar. This might in turn create new empty productions, which are handled in the same way, until there are no empty productions left.
- Removing the unit productions. To remove a unit production $A \to B$, where B is not a terminal, you must add an extra production $A \to \beta$ for each $B \to \beta$ already in the grammar.
- Removing the long productions. Every production $A \to BCD\alpha$ will be replaced by two productions $A \to BA'$ and $A' \to CD\alpha$, where A' is a new category. The second of the new productions might in turn need to be transformed.

Observe that the second and third steps increases the size of the grammar, which in the worst case can become considerably larger.

□

2.3.1 Empty categories

To build the set of empty categories we start with an empty set, and build up the final set of categories by adding new categories whose right-hand side is empty.

```
empties   ::  Grammar c t → Set c
empties (_, _, _, productions)          =  empties′ emptySet
  where empties′ cats  | cats == cats′  =  cats
                       | otherwise      =  empties′ cats′
          where cats′  =  makeSet [ cat | (cat, rhs) ← elems productions,
                                          all (‘elemSet‘ cats) rhs ]
```

Unfortunately it is not possible to use the *limit* function, since we cannot define a suitable *more* function.

Exercise

Define the function that returns the left corners of a sequence of categories.

```
leftCorners  ::  Grammar c t → [ c ] → Set c
```

The left corners of a sequence $A\alpha$ of categories is calculated as follows: We calculate the left corners of the single category A, and if A is an empty category we also add the left corners of α. The left corners of the single category A is calculated by adding A to the union of the left corners of all β, such that $A \to \beta$.

Observe that we have to remember the categories we already have taken care of, to stop us from falling into infinite recursion.

Exercise

Define a function that returns the follow set of a category.

$$\mathit{follow} \; :: \; \mathsf{Grammar}\ c\ t \rightarrow c \rightarrow \mathsf{Set}\ c$$

To calculate the follow set of a category A we can proceed as follows: For every production $B \rightarrow \alpha A \beta$ we take the left corners of the sequence β, and if all categories in that sequence are empty we add the follow set of the category B. Here we also have to remember the categories visited to avoid infinite looping.

Exercise

Define the predicates *cyclic* and *leftRecursive*, saying whether a given category is cyclic or left-recursive.

$$\begin{array}{lll} \mathit{cyclic} & :: & \mathsf{Grammar}\ c\ t \rightarrow c \rightarrow \mathsf{Bool} \\ \mathit{leftRecursive} & :: & \mathsf{Grammar}\ c\ t \rightarrow c \rightarrow \mathsf{Bool} \end{array}$$

□

2.4 Parser combinators

Parser combinators were introduced in 1975 by Burge [2], but it was only after Wadler's paper [44] that people started using them more extensively.

In the last 10 years there has been a lot of research on creating more and more efficient parser combinators [16, 17, 23, 26, 36, 39, 40]. So also in this thesis, where we in chapter 4 introduce some new combinator parsers.

Wadler's original parser was a function from strings to lists of results, where results are pairs of values and the unparsed part of the string.

$$\textbf{type}\ \mathsf{WadlerParser}\ \alpha \;=\; \mathsf{String} \rightarrow [\,(\mathsf{String},\ \alpha)\,]$$

On this parser he added some basic combinators for describing context-free (and other) grammars. His type of parsers will be generalized in this section to the class of combinator parsers, which then can be used to implement any kind of combinator parser. The combinators we introduce can be seen as a simple domain-specific embedded language [14] of parsers.

The class of parsers A combinator parser can be used to parse sequences of input symbols, returning parse results. There are many possible types for the parse function, depending on whether the parser is deterministic or not, and whether we have error reporting or not. As the type of results we will use what has become the de facto standard for combinator parsers, which is a list of values. We will not use any error reporting, but instead argue in the end of this chapter that it is not difficult to add. In some cases one might be interested in parsing just a prefix of the input sentence, and in other cases one is interested in the whole sequence. For this reason, and for the reason that some parsers are more suited for the first kind of recognition, and other parsers for the second, we will have two parse functions in our parser class.

```
class Parser m s | m → s where
  parse           ::  m α → [ s ] → [ ([ s ], α) ]
  parseFull       ::  m α → [ s ] → [ α ]
  parseFull p inp  =  [ a | ([ ], a) ← parse p inp ]
```

In this type class we abstract over the type of the input symbols. This can be done by using the extension of multi-parameter type classes with functional dependencies [18] further described in the introduction, section 1.2.2.

The most obvious choice for the type of the input is a list of terminal symbols, but there are other choices; e.g. a list of sets of terminals (if the lexer is ambiguous) or a word graph from a speech recognizer.[5]

2.5 Context-free combinators

We start with defining combinators which allow us to implement context-free grammars. We do this by introducing type classes for the different combinators.

Monoids First we need a way to introduce more than one production for a category. We do this with the (<+>) combinator, which combines two parsers into one. The meaning is that either of the two parsers are correct, which corresponds to two alternative productions for the same category. We also introduce the always failing parser *zero*, which corresponds to a category without any productions at all. We put both (<+>) and *zero* in the same type class called a monoid. For convenience we also add a combinator *anyof*, which can be defined in terms of the other two.

```
class Monoid m where
  zero   ::  m α
  (<+>)  ::  m α → m α → m α
  anyof  ::  [ m α ] → m α
  anyof   =  foldr (<+>) zero
```

A famous instance of the monoid class is the type of lists, where *zero* corresponds to the empty list, and (<+>) is the same as list concatenation. Another instance is the type of endomorphisms, or functions $\alpha \to \alpha$, where *zero* is the identity function and (<+>) is function composition. Unfortunately the type of sets

[5]A word graph can e.g. be implemented as a connected directed acyclic graph with words on the edges.

cannot be turned into a monoid, since the type of elements has to have an ordering, and we haven't assumed an ordering in the definition of the monoid operations.

Pre-monads We also need a parser that always succeeds, corresponding to an empty production. Since a parser always returns a result, we will use a function that takes a result and gives a parser that always succeeds, returning the given result. This function is the same as the monadic *return*, but since we only want that function and not the monadic bind, we put it into a new class which we call a pre-monad.[6]

```
class PreMonad m where
  return  ::  α → m α
```

Sequencing To get productions having more than one category on the right-hand side, we need a sequencing combinator (<⋆>). This particular combinator was originally defined in [41]. It takes two parsers and combines them in sequence. The result of the combined parser will be the result of the first parser applied to the result of the second. For convenience we also add a limited combinator which throws away the result of the first parser. We call the final class the Sequence class, and to be able to define the limited (⋆>) in terms of (<⋆>) it must also be a premonad.

```
class PreMonad m ⇒ Sequence m where
  (<⋆>)   ::  m (α → β) → m α → m β
  ( ⋆>)   ::  m α → m β → m β
  p ⋆> q  =   (return (λx y → y) <⋆> p) <⋆> q
```

Input symbols Finally we must be able to recognize input symbols, which will correspond to unit productions with a terminal in the right-hand side. So we introduce a function *sym* that takes an input symbol and gives a parser recognizing that symbol. We also introduce two parsers *sat* and *skip*. The *sat* parser recognizes the input symbols for which a given predicate is true, and the *skip* parser succeeds for any input symbol.

```
class Eq s ⇒ Symbol m s | m → s where
  sym   ::  s → m s
  sat   ::  (s → Bool) → m s
  skip  ::  m s
```

As for the Parser class we have to use multi-parameter type classes with functional dependencies to be able to abstract over the type of input symbols. The

[6]This is not possible to implement directly in Haskell, since the Monad class is already defined, but the presentation becomes clearer if we split the Monad class into two classes.

sym and *skip* parsers are definable in terms of *sat*.

$$\begin{array}{lcl} \mathit{sym}\ s & = & \mathit{sat}\ (s ==) \\ \mathit{skip} & = & \mathit{sat}\ (\lambda x \rightarrow \mathsf{True}) \end{array}$$

To define *sat* in terms of *sym* we need a list of all possible input symbols from which we can filter out the ones not satisfying the predicate. And it is impossible to define *sym* or *sat* in terms of *skip* unless the parser is a monad.

Precedence and associativity In this thesis we use the following precedences and associativities for parser combinators, to minimize the use of parentheses.[7]

$$\begin{array}{ll} \textbf{infixr}\ 4 & <:> \\ \textbf{infixl}\ 3 & <\star> \\ \textbf{infixl}\ 3 & \star> \\ \textbf{infixl}\ 2 & <+> \end{array}$$

Recall that a higher value means that the operator binds harder, and **infixl** means left and **infixr** right associativity. This means that $p <\star> q <\star> r$ is the same as $(p <\star> q) <\star> r$ and that $p <:> q <:> r$ is the same as $p <:> (q <:> r)$. Also recall that the monadic combinators have very low precedences in Haskell.

$$\begin{array}{ll} \textbf{infixl}\ 1 & \gg= \\ \textbf{infixl}\ 1 & \gg \end{array}$$

2.6 Parsers and context-free grammars

The combinators given above can be said to be context-free, since they are equivalent to context-free grammars in the following sense.

- Every context-free grammar can be transformed to a combinator parser recognizing the same language.
- Every *non-parametrized* combinator parser can be transformed to a context-free grammar recognizing the same language.

A parametrized parser is a function that calculates a parser from a given argument. We also say that a parser calling a parametrized parser is parametrized itself. It is important that the parser in the second case is not parametrized, which will be clear below.

[7]The (<:>) combinator will be defined later in this chapter.

2.6.1 Parametrized parsers are not context-free

If we allow parsers to be parametrized over, we can recognize languages known not to be context-free. This should not be surprizing, since e.g. a function from integers to parsers can construct a totally new parser for every input, which yields infinitely many parsers, which in turn corresponds to infinitely many nonterminals and/or productions.

The reduplication language The following *redup* parser is a parser recognizing the *reduplication* language $\alpha\alpha$ ($\alpha \in \{a, b\}^*$), which cannot be recognized by any context-free grammar.

```
redup       =     p [ ]
p ts        =     q ts
            <+>   sym 'a' ⋆> p ('a' : ts)
            <+>   sym 'b' ⋆> p ('b' : ts)
q [ ]       =     return ()
q (t : ts)  =     sym t ⋆> q ts
```

The *p* parser builds a list of all the terminals it has seen so far, which is then given to the *q* parser to recognize. Both these parsers are functions depending on a list, which makes them parametrized, and so the *redup* parser is also parametrized.

The $a^n b^n c^n$ language Here is a parser recognizing the language $a^n b^n c^n$, which is also known not to be context-free.

```
abc     =     as 0
as n    =     bs n ⋆> cs n
        <+>   sym 'a' ⋆> as (n + 1)
bs 0    =     return ()
bs n    =     sym 'b' ⋆> bs (n − 1)
cs 0    =     return ()
cs n    =     sym 'c' ⋆> cs (n − 1)
```

2.6.2 Non-parametrized parsers are context-free

If the parser is not parametrized over, it can be transformed into a context-free grammar recognizing the same language.

The transformation consists of two steps, where the first step is to introduce local definitions for each inner application of the five combinators. This is always possible in pure functional languages, because of the law of beta-reduction. Consider the following example parser.

```
p    =   q <⋆> p <+> return (λx → [ x ]) <⋆> sym 'a'
q    =   return (:) <⋆> (sym 'b' <+> zero)
```

We have three applications of ($<\star>$), two applications each of ($<\!+\!>$), *return* and *sym*, and finally one application of *zero*. This gives eight new intermediate parsers, with new unique names.

```
p    =  p_1 <+> p_2
  where p_1  =  q <*> p
        p_2  =  p_3 <*> p_4
        p_3  =  return (λx → [ x ])
        p_4  =  sym 'a'
q    =  q_1 <*> q_2
  where q_1  =  return (:)
        q_2  =  q_3 <+> q_4
        q_3  =  sym 'b'
        q_4  =  zero
```

The second step of the transformation is to create a context-free grammar. Every definition, including the new local definitions, will be transformed to a context-free production rule. The definition will be one of the following,

- **Empty.** The definition $p = zero$ translates to nothing – a nonterminal without any productions.
- **Disjunction.** The definition $p = q <\!+\!> r$ translates to the productions $P \to Q$ and $P \to R$.
- **Unit.** The definition $p = return\ a$ translates to the empty production $P \to \epsilon$.
- **Sequence.** The definition $p = q <\star> r$ translates to the production $P \to Q\ R$
- **Terminal.** The definition $p = sym\ s$ translates to the production $P \to s$.

The nonterminals of the grammar will be the names of all the defined parsers; and the terminals will be all the arguments to the *sym* combinator. Finally we get the following productions for our example parser:

$$\left\{ \begin{array}{llllll} P \to P_1, & P_1 \to Q\ P, & P_3 \to \epsilon & Q \to Q_1\ Q_2, & Q_2 \to Q_4, \\ P \to Q_1, & P_2 \to P_3\ P_4, & P_4 \to \text{'a'}, & Q_2 \to Q_3, & Q_3 \to \text{'a'} \end{array} \right\}$$

2.6.3 Parsers can recognize context-free grammars

Every context-free grammar can be recognized by a corresponding combinator parser, and the creation of the parser is left as an exercise for the reader. The only crucial point is that the grammar must not be left-recursive, so first we have to transform out the possible left-recursion in the grammar. But since this is a standard procedure, we thereby conclude that there is a combinator parser that recognizes the same language as any given context-free grammar.

The reason why the grammar must not be left-recursive is that it can only parse strings using the recursive descent parsing algorithm. And when recursive descent parsing tries to recognize a left-recursive nonterminal X it first has to recognize X, and to recognize X it has to recognize X, and so on ad infinitum…

Exercise

Given a context-free grammar that is not left-recursive, describe a procedure to transform the grammar into a combinator parser returning the unit element ().

Exercise

Give another procedure to transform the grammar into a parser returning a parse tree.

□

2.7 Monadic combinators

Most definitions of parser combinators use a monadic framework. This is very handy, there is a nice syntactic sugar in the **do**-notation, and one can exploit a richer language than mere context-free languages.

The main difference to the context-free combinators is that we use another combinator for sequencing. This new sequencing combinator will be the monadic bind ($\gg=$), which takes as arguments a parser and a function from results to parsers (called a *continuation*), and returns a new parser. The result of the first parser decides what the parser to use next will be. For convenience we also introduce a more limited sequencing combinator, where the result of the first parser is ignored.

```
class PreMonad m ⇒ Monad m where
  (≫=)    ::  m α → (α → m β) → m β
  (≫)     ::  m α → m β → m β
  p ≫ q   =   p ≫= (λx → q)
```

In Haskell there is syntactic sugar that can be used instead of the monadic bind – the **do**-notation. This makes monadic programs (such as parsers) easier to read. But we will not have to use the **do**-notation in this thesis, since the grammars are not that complicated.

With the monadic bind it is simple to define parsers that recognize non context-free languages. This is clear since the second argument to ($\gg=$) can invoke any Haskell function, and so the strength of the language becomes Turing-complete.

The reduplication language Here is a monadic parser to recognize the reduplication language $\alpha\alpha$ ($\alpha \in \{a,b\}^*$).

```
redup  =    ps >>= λxs → ps >>= λys →
            if (xs == ys) then return () else zero
p      =    sym 'a' <+> sym 'b'
ps     =    return [ ]
      <+>   (p >>= λx → ps >>= λxs → return (x : xs))
```

The $a^nb^nc^n$ language This is a monadic parser that recognizes the language $a^nb^nc^n$.

```
abc    =    ps 'a' >>= λa → ps 'b' >>= λb → ps 'c' >>= λc →
            if a == b ∧ b == c then return () else zero
ps s   =    return 0
     <+>    (sym s >> ps s >>= λn → return (n + 1))
```

2.8 Deriving parser combinators

The Functor class transforms a parser into a parser of another type, by applying a given function to each of the results.

```
class Functor m where
  fmap  ::  (α → β) → m α → m β
```

It is well-known how to turn a monad into a functor, and it is already implemented in the standard Monad library as the *liftM* function.[8]

```
instance Monad m ⇒ Functor m where
  fmap f p   =   liftM f p
             =   p >>= λa → return (f a)
```

It is also easy to turn a context-free parser into a functor, by sequencing an empty parser with the given parser.

```
instance Sequence m ⇒ Functor m where
  fmap f p   =   return f <⋆> p
```

Since the monadic bind (>>=) is more general than the sequence (<⋆>), we should be able to define the latter in terms of the former. And as a matter of fact sequencing is already defined in the Monad library as the *ap* combinator.

[8]Sometimes we will write several definitions of the same function below each other, which only means that any of the definitions is suitable.

The definition for $(\star\!>)$ becomes even simpler, since it is equivalent to the $(\gg)$ combinator.

```
instance Monad m ⇒ Sequence m where
  p <⋆> q  =  ap p q
           =  p >>= λf → fmap f q
  p ⋆> q   =  p >> q
```

2.8.1 Other derived combinators

Recognizers Sometimes we are not interested in a result, but only if the parse succeeds or not. We then use the dummy parse result () to indicate that the parse succeeds, and sometimes call a parser returning () a recognizer. We also introduce the dummy parser *success* which simply succeeds, returning the dummy result ().

```
success  ::  PreMonad m ⇒ m ()
success  =   return ()
```

A common parametrized parser is the $many_0$ combinator, which recognizes any number of the given parser in sequence, including the empty sequence.

```
many₀        ::  (Monoid m, Sequence m) ⇒ m α → m ()
many₀ p      =   ps
  where ps   =   success <+> p ⋆> ps
```

It is also convenient with a parser that recognizes a sequence of input symbols. We define this as a combinator $syms_0$.

```
syms₀            ::  (Sequence m, Symbol m s) ⇒ [ s ] → m ()
syms₀ [ ]        =   return ()
syms₀ (s : ss)   =   sym s ⋆> syms ss
```

List returning parsers Most often we are interested in results, and for that reason we define some useful combinators with lists as results. The (<:>) combinator is a lifted version of the (:) constructor on lists.

```
(<:>)      ::  Sequence m ⇒ m α → m [ α ] → m [ α ]
p <:> ps   =   return (:) <⋆> p <⋆> ps
           =   fmap (:) p <⋆> ps
```

We also have the *many* combinator, which is like $many_0$ but it returns a list of all the recognized results.

```
many         ::  (Monoid m, Sequence m) ⇒ m α → m [ α ]
many p       =   ps
  where ps   =   return [ ] <+> p <:> ps
```

Finally the *syms* combinator is like $syms_0$, but it returns the recognized sequence.

```
syms          ::  (Sequence m, Symbol m s) ⇒ [s] → m [s]
syms []        =  return []
syms (s : ss)  =  sym s <:> syms ss
```

Exercise

Define the *redup* parser and the *abc* parser from section 2.6.1 using the new derived combinators.

□

2.9 Error reporting combinators

In this thesis we do not consider how to report errors while parsing, which is a very interesting subject of its own. There are many different views on how to do error reporting, and most of them are fairly straightforward to add to any given parser type. Here is a simple way, and by far not the only, to add error reporting to a combinator parser.

- We have to change the result type from lists of values to something that can hold information about parse errors. This is just an example of how the new type can look like.

  ```
  type Result s α  =  Either (Error s) [α]
  ```

  The type of errors can e.g. be the position of the error, what the parser was expecting, and what the parser did find. The position can e.g. be an index in the input sequence, and the other two can be sets of input symbols describing the token that was expected and the token that was found.

  ```
  type Error s     =  (Position, Expected s, Found s)
  type Position    =  Int
  type Expected s  =  Set s
  type Found s     =  Set s
  ```

- And when we change the result type, we have to reflect that in the types of the functions in the Parser class.

  ```
  class Parser m s | m → s where
    parse      ::  m α → [s] → Result s ([s], α)
    parseFull  ::  m α → [s] → Result s α
  ```

- Finally we have to do changes to the definitions of the parser combinators, which is left as an exercise for the reader. The main thing there is to increase the position number whenever a token is being read.

Exercise

Add error reporting to each of the combinator parsers defined in chapter 3 and 4.

Exercise

In [26, 41] the authors add an extra combinator (<?>) so that the programmer can give explicit names to different subparsers. This makes it possible to make more readable error messages – the *parse* function can reply that it was expecting a "number" instead of just saying that it wanted any of the characters "0", "1", . . . , "9". Extend the error reporting parsers from the last exercise with this useful combinator. It might be necessary to read [26] first, to understand how the (<?>) combinator works.

□

2.9.1 Error correcting combinators

Swierstra et al have given examples of how to do simple error recovery, to be able to continue the parsing process and thereby hopefully catch more than one parse error at the time [39, 40, 41]. This proves to be more difficult to add as a feature to a given combinator parser, and in this thesis we do not pursue this path any further.

2.10 Discussion

Sets and maps There are of course different implementations of sets and finite maps, and [32] is a good source for functional implementations of these data structures.

The implementations in the appendix are based on ordered lists, which might sound too simple for efficient implementations. But in fact it is only the lookup function for finite maps that can (and should) be implemented in a more efficient way.

Complexity of set operations The complexity of the *elemSet* function is of course linear in the size of the set, and it is a simple task to convert the ordered list to a binary search tree to get logarithmic time lookup. But since set membership is not used in any important functions, it is not necessary to complicate the type.

Set union and difference is used much more extensively. The (<++>) and (<-->) operations are also linear time $O(n)$, in the size n of the final set. The *union* of a list of sets consists of a number m of applications of (<++>), so the time complexity is $O(nm)$, where n is the size of the final union and m is the number of sets to union.

The *limit* function is quadratic in the size of the final fix-point set. Suppose that *limit more start* has n elements. According to the definition in section 2.2.1, the *more* function will be called exactly once for each added element. And since the result of the application of *more* is a subset of the final set, it will at most give n new elements. This means that the final set is calculated as n unions of sets with $O(n)$ elements, which gives us $O(n^2)$. This result is true only if the *more* function does not have complexity more than linear in n, otherwise the complexity will be $O(nm)$, where $O(m)$ is the complexity of *more*.

Efficient lookup for finite maps In several chapters we will use finite maps extensively, and especially the lookup function (?). This means that the linear time lookup should be improved upon. It suffices to simply pair the ordered association list with a balanced binary search tree which is built directly from the list. All our implementations will first build the map using *mergeWith*, $(|\!\rightarrow)$, *makeMapWith*, et cetera; and then lookup elements when the map is built. This means that the linear time conversion from association lists to binary trees will only be applied once.

Grammars There are of course also different possible implementations of context-free grammars. Perhaps the most straightforward one is to use the Either type for the set $V = N \cup \Sigma$, which would give a different type for the productions in the grammar.

$$\textbf{type}\ \mathsf{Production}\ c\ t \quad = \quad (c,\ [\,\mathsf{Either}\ c\ t\,])$$

The type of grammars would not need the *terminal* function transforming a terminal to a set of non-terminals. This approach would be more in line with the standard definitions of context-free grammars, but we have chosen another path.

One reason for this is that the implementations of the parsing algorithms become simpler and more elegant with our definitions. Another reason is that the *terminal* function can be thought of as a tokenizer/morphological analyzer, which anyhow will be implemented in a real application.

Combinator parsers The class hierarchy of the parser combinators might seem complicated, and there are of course simpler ways of defining parsers. One important reason for the hierarchy is that we want to be able to define both context-free and monadic parsers within the same framework.

In many implementations of combinator parsers (such as [17, 23, 36, 44]), the authors treat the problems of space leaks and unwanted ambiguities by introducing combinators for pruning the search space. They are called "cut", "xor" or "deterministic choice", but they all amount to approximately the same behaviour. The relations between these approaches, and their behaviour on different parser types, is a very interesting subject which we do not cover in this thesis.

Derived combinators The derived combinators in section 2.8.1 are only small examples of what can be done. For more elaborate examples we refer to the literature, e.g. [17, 26, 40].

Chapter 3

Existing Parser Combinators

In this chapter we define some standard parser combinators, which are already defined in the literature. First we define different backtracking parsers, which are all descended from Wadler's original parser. Then we shift focus to breadth-first searching parsers, which means variants of the stream processor parser.

3.1 Backtracking Parser Combinators

Almost all combinator parsers that have been proposed are based on backtracking. Furthermore, these parsers are all based on the standard "function-from-strings-to-lists-of-results" parser, first suggested by Wadler [44].

3.2 The standard parser

This is the basic parser from which every other backtracking parser derives. It has been intensively studied ever since Wadler's first papers [44, 45]. The standard parser is just a reflection of the type of the *parse* function, using lists of input symbols.

```
newtype Standard s α  =  Std ([s] → [([s], α)])
```

Since a list already is a monoid, we just lift the zero and choice to functions.

```
instance Monoid (Standard s) where
  zero               =  Std (λinp → [ ])
  Std p <+> Std q    =  Std (λinp → p inp ++ q inp)
```

The *return* parser does not consume any input, and returns the argument.

```
instance PreMonad (Standard s) where
  return a   =   Std (λinp → [(inp, a)])
```

To turn it into a monad, we have to define the (≫=) combinator. This is done by applying the first parser to the input, getting a list of remaining inputs and results, then applying the continuation to each of the results, and applying the resulting parser to the remaining input; finally concatenating all the possible results.

```
instance Monad (Standard s) where
  Std p ≫= k   =   Std (λinp → concat [ q inp′ | (inp′, a) ← p inp,
                                                   let Std q = k a ])
```

The *sat* parser only have to check if the next input symbol satisfies the given predicate, and return the remaining input together with the matched symbol as result.

```
instance Eq s ⇒ Symbol (Standard s) s where
  sat p   =   Std sat′
    where sat′ (s : inp) | p s   =   [(inp, s)]
          sat′ _                 =   [ ]
```

Since the standard parser is a parsing function by itself, we use that function as the parse function.

```
instance Parser (Standard s) s where
  parse (Std p) inp   =   p inp
```

3.3 Continuation based combinator parsers

The standard parser is not the best way to implement a backtracking parser. There are two reasons for this: list concatenation and list concatenation. Both in the definition of (≫=) and in the definition of (<+>) we have used list concatenation; and we tackle the two problems in order.

3.3.1 The continuation monad transformer

The first list concatenation problem is that the definition of bind (≫=) for the list monad requires a concatenation of a list of lists, which can be inefficient. The problem is that the sequencing combinators associate to the left, which means that the corresponding list concatenations also will associate to the left. And left-associative list concatenating is expensive, since lists themselves are right-associative.[1] Another problem is that the standard parser will create lots of intermediate lists which will immediately be garbage collected, and therefore slow down the parsing. The solution to these problems is to use a continuation monad transformer. This is one of the monad transformers by Liang, Hudak and Jones [27].

The continuation transformer is a function that takes as argument the future – a description of what we want to do with the result of the parser – and applies the future to the result. The result of the application can be of any particular type, which means that we must use the Haskell extension of existential quantification in types [9], further described in the introduction, section 1.2.2, to be able to make it an instance of the parser classes.

$$\textbf{newtype}\ \mathsf{ContTrans}\ m\ \alpha \quad = \quad \mathsf{CT}\ (\forall \beta.\ (\alpha \to m\ \beta) \to m\ \beta)$$

The nice thing about a continuation transformer is that it is a monad regardless of whether the underlying type is a monad or not. The *return* function just applies the future to the result.

$$\begin{aligned}&\textbf{instance}\ \mathsf{PreMonad}\ (\mathsf{ContTrans}\ m)\ \textbf{where}\\ &\quad \mathit{return}\ a \quad = \quad \mathsf{CT}\ (\lambda \mathit{fut} \to \mathit{fut}\ a)\end{aligned}$$

The bind combinator waits for a result a for the first parser p. Then it applies the continuation k to the result a to get a new parser q, which is applied to the future.

$$\begin{aligned}&\textbf{instance}\ \mathsf{Monad}\ (\mathsf{ContTrans}\ m)\ \textbf{where}\\ &\quad \mathsf{CT}\ p \gg\!= k \quad = \quad \mathsf{CT}\ (\lambda \mathit{fut} \to p\ (\lambda a \to \textbf{let}\ \mathsf{CT}\ q = k\ a\ \textbf{in}\ q\ \mathit{fut}))\end{aligned}$$

If the underlying type is a monoid, we can lift the monoid functions to the continuation monad.

$$\begin{aligned}&\textbf{instance}\ \mathsf{Monoid}\ m \Rightarrow \mathsf{Monoid}\ (\mathsf{ContTrans}\ m)\ \textbf{where}\\ &\quad \mathit{zero} && = \quad \mathsf{CT}\ (\lambda \mathit{fut} \to \mathit{zero})\\ &\quad \mathsf{CT}\ p <\!\!+\!\!> \mathsf{CT}\ q && = \quad \mathsf{CT}\ (\lambda \mathit{fut} \to p\ \mathit{fut} <\!\!+\!\!> q\ \mathit{fut})\end{aligned}$$

To be able to parse a sentence with the continuation monad, the underlying type needs to be a parser, as well as a premonad. The underlying *return* function is used as the initial future passed to the continuation parser.

$$\begin{aligned}&\textbf{instance}\ (\mathsf{PreMonad}\ m,\ \mathsf{Parser}\ m\ s) \Rightarrow \mathsf{Parser}\ (\mathsf{ContTrans}\ m)\ s\ \textbf{where}\\ &\quad \mathit{parse}\ (\mathsf{CT}\ p)\ \mathit{inp} \quad = \quad \mathit{parse}\ (p\ \mathit{return})\ \mathit{inp}\end{aligned}$$

The only thing left to define now is the instance for the Symbol class. But to avoid inefficiencies, we postpone that definition until we know what type constructor we are going to apply the continuation transformer to.

[1] The left-associative concatenation $((\ldots(xs_1 +\!\!+ xs_2) +\!\!+ \ldots) +\!\!+ xs_n)$ is quadratic in n; while the right-associative concatenation $(xs_1 +\!\!+ (\ldots +\!\!+ (xs_{n-1} +\!\!+ xs_n)\ldots))$ is linear.

Exercise

This is a perfectly functioning definition of the *sat* parser in terms of the underlying type's *sat* parser.

```
instance (Monad m, Symbol m s) ⇒ Symbol (ContTrans m) s where
  sat p   =   CT (λfut → sat p >>= fut)
```

In what way is this definition inefficient?

□

3.3.2 The standard continuation parser

To get a working continuation parser we apply the continuation transformer to the standard parser.

```
type StandardCont s α   =   ContTrans (Standard s) α
```

Now we can define a version of the *sat* parser, which doesn't use the *sat* of the standard parser, by applying the future to the next input symbol.[2]

```
instance Symbol (StandardCont s) s where
  sat p    =    CT (λfut → Std (sat' fut))
    where sat' fut (s : inp)  | p s   =   let Std p = fut s in p inp
          sat' _  _                   =   [ ]
```

3.3.3 The endomorphism parser

The second list concatenation problem of the standard parser lies in the definition of alternation – we use list concatenation to join the results together. The problems are much the same as described for the first list concatenation problem, but the solution is slightly different. The solution is to use a function from lists to lists instead of just lists. This is sometimes called an endomorphism over lists, and is used in e.g. the language definition of Haskell, in the ReadS and ShowS types.

An endomorphism is a monoid, just as a list is. The *zero* is the identity function, and the (<+>) is function composition. This means that we can define a more efficient variant of the standard parser, by using an endomorphism instead of a

[2]To be correct, we need to define the StandardCont type with a **newtype** declaration, but this is trivial.

list as result type.

$$\textbf{newtype}\ \mathsf{StandardEndo}\ s\ \alpha \quad = \quad \mathsf{StdE}\ ([\,s\,] \rightarrow [\,([\,s\,],\ \alpha)\,] \rightarrow [\,([\,s\,],\ \alpha)\,])$$

$$\begin{array}{lll}
\textbf{instance}\ \mathsf{Monoid} & (\mathsf{StandardEndo}\ s) & \textbf{where}\ldots \\
\textbf{instance}\ \mathsf{PreMonad} & (\mathsf{StandardEndo}\ s) & \textbf{where}\ldots \\
\textbf{instance}\ \mathsf{Monad} & (\mathsf{StandardEndo}\ s) & \textbf{where}\ldots \\
\textbf{instance}\ \mathsf{Symbol} & (\mathsf{StandardEndo}\ s)\ s & \textbf{where}\ldots \\
\textbf{instance}\ \mathsf{Parser} & (\mathsf{StandardEndo}\ s)\ s & \textbf{where}\ldots
\end{array}$$

Unfortunately this parser will be very inefficient, since the definition for ($\gg=$) becomes awkward.

Exercise

Write the instance definitions of the parser classes for the endomorphism parser.

Exercise

Abstract away the result type from the standard parser to get the parser transformer StandardAbs.

$$\textbf{newtype}\ \mathsf{StandardAbs}\ m\ s\ \alpha \quad = \quad \mathsf{StdA}\ ([\,s\,] \rightarrow m\ ([\,s\,],\ \alpha))$$

Write the instance declarations for this parser. What classes must the abstracted constructor m be instances of?

□

3.3.4 The endomorphism continuation parser

Finally we can use the standard endomorphism parser as the base type for the continuation transformer, to get a backtracking parser which doesn't use list concatenation at all.

$$\textbf{type}\ \mathsf{StandardEndoCont}\ s\ \alpha \quad = \quad \mathsf{ContTrans}\ (\mathsf{StandardEndo}\ s)\ \alpha$$

All instance declarations for the continuation transformer will of course remain the same. It is only the symbol parser that has to be tailor-made for this parser.

Exercise

Define the Symbol instance for the endomorphism contiuation parser.

□

Versions of this final backtracking parser have been used in the works of Röjemo [36], Koopman and Plasmeijer [23] and others, to build efficient backtracking parsers. The authors tackle the problems of backtracking discussed in section 3.5, by introducing yet more combinators that can be used to prune the search tree. Such a solution is more of ad-hoc nature and is not the path we pursue in this thesis, so we leave these pruning, or "cutting", combinators as they are.

3.4 The stack continuation transformer

Doaitse Swierstra has in some papers introduced an efficient parser which has a very complicated structure [39, 40]. This parser can be decomposed into three parts: error correction, a trie structure and an underlying parser. The error correction is not part of this thesis. The trie structure will be discussed in the next chapter, in section 4.4.

From the underlying parser one can extract a new kind of continuation transformer, which we will call the stack continuation transformer.

$$\textbf{newtype}\ \mathsf{StackTrans}\ m\ \alpha \quad = \quad \mathsf{ST}\ (\forall \beta\gamma.\ (\beta \to m\ \gamma) \to (\alpha \to \beta) \to m\ \gamma)$$

The type is similar to the ordinary continuation transformer, but the parser takes an extra argument apart from the future. This extra argument is a function $\alpha \to \beta$ and will work as a stack of functions, translating the result to a value accepted by the future. To use two continuations in this way makes it possible to make use of sharing in a better way than the standard continuation parser can.

The stack transformer is a pre-monad, where the future is applied to the result of applying the stack to the value.

$$\begin{aligned}&\textbf{instance}\ \mathsf{PreMonad}\ (\mathsf{StackTrans}\ m)\ \textbf{where}\\ &\quad \mathit{return}\ a \quad = \quad \mathsf{ST}\ (\lambda \mathit{fut}\ \mathit{stack} \to \mathit{fut}\ (\mathit{stack}\ a))\end{aligned}$$

The stack transformer is also an instance of Sequence. The $(<\star>)$ combinator takes two parsers as arguments. The second is applied to the future, to become the future for the first parser, similar to the ordinary continuation parser.

$$\begin{aligned}&\textbf{instance}\ \mathsf{Sequence}\ (\mathsf{StackTrans}\ m)\ \textbf{where}\\ &\quad \mathsf{ST}\ p <\star> \mathsf{ST}\ q \quad = \quad \mathsf{ST}\ (\lambda \mathit{fut}\ \mathit{stack} \to p\ (q\ \mathit{fut})\ (\mathit{stack}\ \cdot))\end{aligned}$$

If f is the value of the first parser and a is the value of the second, the resulting value should be $f\ a$. And if we apply the stack $(\mathit{stack}\cdot)$ to f and a, we get what we want – i.e. the resulting value will be stacked.

$$(\mathit{stack}\ \cdot)\ f\ a = (\mathit{stack} \cdot f)\ a = \mathit{stack}\ (f\ a)$$

The monoid instance is just lifted from the underlying type.

$$\begin{aligned}&\textbf{instance}\ \mathsf{Monoid}\ m \Rightarrow \mathsf{Monoid}\ (\mathsf{StackTrans}\ m)\ \textbf{where}\\ &\quad \mathit{zero} \quad &&= \quad \mathsf{ST}\ (\lambda \mathit{fut}\ \mathit{stack} \to \mathit{zero})\\ &\quad \mathsf{ST}\ p <\!+\!> \mathsf{ST}\ q \quad &&= \quad \mathsf{ST}\ (\lambda \mathit{fut}\ \mathit{stack} \to p\ \mathit{fut}\ \mathit{stack} <\!+\!> q\ \mathit{fut}\ \mathit{stack})\end{aligned}$$

Similar to the continuation transformer, we get an underlying parser by applying the parser to the initial future (which is *return*) and the initial stack (which will be the identity function).

$$\begin{aligned}&\textbf{instance}\ (\mathsf{PreMonad}\ m,\ \mathsf{Parser}\ m\ s) \Rightarrow \mathsf{Parser}\ (\mathsf{StackTrans}\ m)\ s\ \textbf{where}\\ &\quad \mathit{parse}\ (\mathsf{ST}\ p)\ \mathit{inp} \quad = \quad \mathit{parse}\ (p\ \mathit{return}\ \mathit{id})\ \mathit{inp}\end{aligned}$$

3.4.1 Comparing the stack continuation with the standard continuation

If we unfold the definitions for the standard continuation, we see that sequencing can be defined as follows.

$$\mathsf{CT}\ p <\!\star\!> \mathsf{CT}\ q \quad = \quad \mathsf{CT}\ (\lambda fut \rightarrow p\ (\lambda f \rightarrow q\ (fut \cdot f)))$$

We then notice that the future given to p is a lambda-abstraction over the f. This in turn breaks the sharing of the q parser, because the q parser with its continuation has to be rebuilt for every application of the sequence.

If we look at the sequence definition for the stack continuation, we see that the lambda-abstraction is missing from the future. This means that q can be shared and doesn't have to be rebuilt for every application.

3.4.2 The standard stack continutation parser

If we apply the stack transformer to the standard parser we get a working stack continuation parser.

$$\textbf{type}\ \mathsf{StandardStack}\ s\ \alpha \quad = \quad \mathsf{StackTrans}\ (\mathsf{Standard}\ s)\ \alpha$$

The definition of the Symbol instance will be very similar to the definitions for the StandardCont parser, we just add the extra stack argument to the helper function.

```
instance Symbol (StandardStack s) s where
  sat p    =   ST (λfut stack → Std (sat′ fut stack))
    where sat′ fut stack (s : inp) | p s  =  let Std p = fut (stack s) in p inp
          sat′ _   _     _                =  [ ]
```

Exercise

Implement the StandardEndoStack parser by applying the stack transformer to the endomorphism parser.

□

3.5 Breadth-first searching parsers

3.5.1 Problems with backtracking

There is a memory problem with the backtracking parsers mentioned in the last section. When we have a choice in the grammar, the parser must hold on to the input at that point until it knows whether the first parser succeeds or not, because it must be able to try the second parser. This gives rise to a space leak – it can make a grammar that could be parsed in constant space to grow linearly in the size of the input.

One solution to this problem is to introduce extra combinators to cut off the search when we know where the solution is, as we briefly described above. Another solution, which is the path we pursue here, is to change from depth-first search to breadth-first search.

3.5.2 The stream processor parser

Koen Claessen [5] has discovered a parser which parses the input in parallel, or breadth-first, instead of depth-first as backtracking parsers do. This makes it possible for the compiler to discard the previous input, thus not introduce the possible space leak introduced by backtracking parsers. The parser in this section is equivalent to one implementation of the stream processor in the graphics library Fudgets [3], hence its name.

We start with a datatype of character streams, which is an extension of the standard list type to also support reading from a stream of characters. So we have a cons and a nil from the list type, and a shift that reads one symbol from the input stream.

```
data Stream s α  =  Shift (s → Stream s α)
                 |  α ::: Stream s α
                 |  Nil
```

The Nil will of course serve as the *zero* of the monoid, and the alternation is an extension of list concatenation. It collects all possible results until both streams want to shift, and then it shifts simultaneously for both.

```
instance Monoid (Stream s) where
  zero                    =  Nil

  Nil        <+> bs       =  bs
  as         <+> Nil      =  as
  (a ::: as) <+> bs       =  a ::: (as <+> bs)
  as         <+> (b ::: bs) =  b ::: (as <+> bs)
  Shift f    <+> Shift g  =  Shift (λs → f s <+> g s)
```

The stream is also a premonad, where the *return* function creates a list with one element, just as for ordinary lists.

```
instance PreMonad (Stream s) where
  return a  =  a ::: Nil
```

To turn the stream type into a monad we pattern match on the first argument.

```
instance Monad (Stream s) where
  Shift f     >>= k  =  Shift (λs → f s >>= k)
  (a ::: as)  >>= k  =  k a <+> (as >>= k)
  Nil         >>= k  =  Nil
```

The stream is also an instance of the Symbol class, with a very simple definition of the *skip* parser.

```
instance Symbol (Stream s) s where
  skip   =  Shift return
  sat p  =  Shift (λs → if p s then return s else zero)
```

To parse a stream we collect all possible results together with what's left of the input until the stream is empty. On a shift we apply the shift function to the next input symbol.

```
instance Parser (Stream s) s where
  parse (Shift f) (s : inp)  =  parse (f s) inp
  parse (a ::: p)  inp       =  (inp, a) : parse p inp
  parse _          _         =  [ ]
```

We can also define the *parseFull* function in the obvious way, which will improve the memory behaviour slightly.

```
parseFull (Shift f) (s : inp)  =  parseFull (f s) inp
parseFull p          [ ]       =  collect p
   where collect    (a ::: p)  =  a : collect p
         collect     p         =  [ ]
parseFull (_ ::: p)  inp       =  parseFull p inp
parseFull _          _         =  [ ]
```

3.5.3 The stream continuation parser

Unfortunately the stream processor parser suffers from the same disadvantages as the standard parser – the definition of ($\gg=$) is inefficient. To get efficient sequencing for the stream processor, we wrap it into a continuation.

```
type StreamCont s α  =  ContTrans (Stream s) α
```

The final thing we have to do is to define the Symbol class in terms of the underlying stream datatype. The *sat* parser needs an input symbol, which means that we have a Shift applying the future to an input symbol satisfying the predicate.

```
instance Symbol (StreamCont s) s where
  sat p  =  CT (λfut → Shift (λs → if p s then fut s else zero))
```

Exercise

The *skip* parser has a very nice definition for this parser. How does the definition look like?

□

3.5.4 The stream stack continuation parser

It is also possible to combine the stream parser with the stack transformer.

$$\textbf{type}\ \mathsf{StreamStack}\ s\ \alpha \quad = \quad \mathsf{StackTrans}\ (\mathsf{Stream}\ s)\ \alpha$$

And the Symbol instance is similar to the definition above, with an extra argument for the stack.

$$\begin{array}{l} \textbf{instance}\ \mathsf{Symbol}\ (\mathsf{StreamStack}\ s)\ s\ \textbf{where} \\ \quad sat\ p \quad = \quad \mathsf{ST}\ (\lambda fut\ stack \rightarrow \mathsf{Shift}\ (\lambda s \rightarrow \textbf{if}\ p\ s\ \textbf{then}\ fut\ (stack\ s)\ \textbf{else}\ zero)) \end{array}$$

3.6 Discussion

Compiler optimizations Today's compilers are very good at optimizing away inefficiencies in program code. One of the more efficient optimization strategies is deforestation, which can be used to get rid of intermediate lists [10]. This suggests that the continuation transformers might not be the best optimization in all cases. And in fact, in some cases the continuation transformers does not give much improvement compared to the standard list parser. But in most cases the transformers give better results, which of course is what we hoped for.

The stack continuation The stack continuation transformer was introduced by Swierstra [39, 39], but it was hidden inside the type of his parser. We have extracted the transformer, and realized that it can be applied to more base parsers than Swierstra uses.

A stack continuation parser is efficient when parsing sequences $p <\star> q$, because it exploits sharing while parsing. The standard continuation parser on the other hand does not exploit sharing, which means that it has to rebuild parsers many times while parsing.

The drawback with the stack continuation is that it is not possible to define the monadic ($\gg\!=$), which means that it has a much more restricted use.

The stream parser The advantage and disadvantage of the stream processor parser is that it is breadth-first instead of depth-first. For certain deterministic grammars which expect large input, e.g. grammars for programming languages, the problem of holding on to previous input is a big problem. Since the stream parser searches breadth-first, the old input can immediately be discarded and garbage-collected. This can lead to a space complexity which is constant instead of linear in the size of the input.

The problem with the stream parser is if the grammar is not left-factorized, i.e. if we have to parse different alternatives in parallel. Then we get the same kind of memory problem that breadth-first search is known to have. We trade one memory problem for another, and it depends on the application if we win or lose.

Chapter 4

Left-factorizing Parser Combinators

In this chapter we define a new class of combinator parsers, building on the data structure of tries, or letter trees. One can see these parsers as generalizations of finite automata, or as simplifications of the parser by Doaitse Swierstra [39, 41].

4.1 Left-factorization

Unfortunately, the stream processor parser described in the last chapter still has one problem left. It is possible to have an alternation in the parser where both choices start with the same input symbol. Then the stream processor parser will test that input symbol twice, once for each choice.

As an example we can take a simple *number* parser, recognizing any of the strings "`three`", "`four`", "`thirty`" and "`forty`".

```
number  =   syms "three"  *> return 3
       <+>  syms "thirty" *> return 30
       <+>  syms "four"   *> return 4
       <+>  syms "forty"  *> return 40
```

If the string to be recognized is "forty", the stream processor parser checks the first letter four times – twice to check for a t, and twice to check for an f – and the second and third letter will be checked twice each – one for the string "four", and one for the string "forty". This is clearly inefficient, and there is a standard solution to this problem: left-factorize the grammar. This means transforming the *number* parser into the equivalent parser *number′*, where we will have a minimal number of tests of input symbols.

```
number'  =   syms "th" ⋆> (syms "ree"   ⋆> return 3   <+>
                           syms "irty"  ⋆> return 30)
        <+>  syms "fo" ⋆> (syms "ur"    ⋆> return 4   <+>
                           syms "rty"   ⋆> return 40)
```

It is always possible to left-factorize a grammar by hand, to make the parser more efficient. Unfortunately this can be a quite involved process, yielding a grammar which is much more difficult to read and understand.

Another solution is to use the data structure of tries, or letter trees, to implement a breadth-first searching parser that left-factorizes the grammar automatically.

4.2 Trie structures

Trie structures, or letter trees, are widely used to represent sets of sequences with efficient lookup – the time to lookup an element can be as fast as linear in the length of the sequence, if we use hash tables to lookup a single element. Tries are also space-efficient, since a prefix for many different sequences is stored only once.

In [32] there is a description on how to implement tries in a functional language. That description is more general than the implementation in this chapter, but they are otherwise very similar.

A trie is a tree-shaped deterministic finite automaton. That it is tree-shaped means that there are no cycles, and that sub-tries cannot be shared between nodes. That it is deterministic means that the trie is left-factorized completely. Usually one draws a trie from left to right, with circles as the nodes of the automaton and marked circles as final nodes, such as figure 4.1.

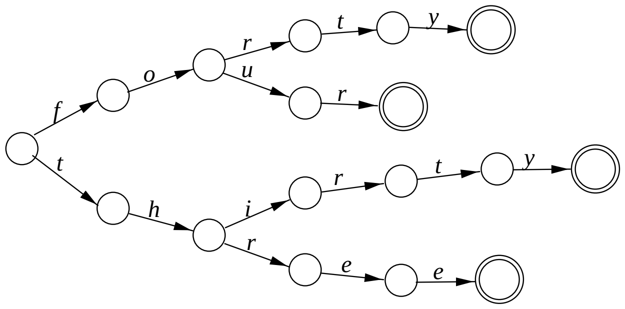


Figure 4.1: A trie of the *number* parser.

Since we are working within a lazy programming language, we can define an infinite trie – only the parts that are needed will be evaluated. Lazy functional languages also have the possibility of sharing common substructures. In the case of tries this means that the trie can be stored as a directed graph, where common sub-tries are shared. The graph can even be cyclic, which means we can represent an infinite trie in finite memory.

For the representation of the edges in the trie we will use finite maps from input symbols to tries. An example implementation of finite maps is in the appendix, implemented as ordered association lists.

4.2.1 The Trie Parser

A trie can be implemented as a kind of list, where the empty list is replaced by a finite map pointing to a new trie. The elements in this "trie list" represent the results of the corresponding node in the trie. This representation is similar to the Stream data type in the previous section, but we won't need any Nil constructor since this can be accomplished with the empty map.[1]

```
data Trie s α  =  Shift (Map s (Trie s α))
               |  α ::: Trie s α
```

The *zero* is simply the empty map, and to combine two tries we just collect all results and then merge the final maps together.

```
instance Ord s ⇒ Monoid (Trie s) where
  zero                           =  Shift emptyMap

  (a ::: p)     <+> q            =  a ::: (p <+> q)
  p             <+> (b ::: q)    =  b ::: (p <+> q)
  Shift pmap <+> Shift qmap      =  Shift (mergeWith (<+>) pmap qmap)
```

A trie is also a premonad, just as the stream processor, where we just take the result followed by the *zero* trie.

```
instance Ord s ⇒ PreMonad (Trie s) where
  return a   =   a ::: zero
```

A trie can even be turned into a monad by pattern matching on the first argument. This definition is very similar to the definition of (≫=) for lists. If we have a result *a*, we apply the continuation *k* to *a* and merge it with the bind for the rest of the trie. If we have reached the end of the "trie list" we have a finite map, and we map the continuation on each of its values.

```
instance Ord s ⇒ Monad (Trie s) where
  (a ::: p)      ≫= k   =   k a <+> (p ≫= k)
  Shift pmap ≫= k   =   Shift (mapMap (≫=k) pmap)
```

[1]To simplify the presentation, we will abuse the Haskell standard and overload the constructor functions Shift, (:::) and others, for each parser in this chapter.

To read a specific input symbol is just to make a shift on that symbol, returning the result.

```
instance Ord s ⇒ Symbol (Trie s) s where
  sym s  =  Shift (s |→ return s)
```

Finally, to parse we just go through the trie much in the same way as we did with the stream processor parser. When we have a shift, we lookup the corresponding input symbol in the finite map to get a new trie to continue with.

```
instance Ord s ⇒ Parser (Trie s) s where
  parse (a ::: p)       inp          =  (inp, a) : parse p inp
  parse (Shift _)       [ ]          =  [ ]
  parse (Shift pmap)    (s : inp)    =  case pmap ? s of
                                          Just p   → parse p inp
                                          Nothing → [ ]

  parseFull p                 [ ]         =  collect p
    where collect             (a ::: p)   =  a : collect p
          collect             _           =  [ ]
  parseFull (_ ::: p)         inp         =  parseFull p inp
  parseFull (Shift pmap)      (s : inp)   =  case pmap ? s of
                                               Just p   → parseFull p inp
                                               Nothing → [ ]
```

With this implementation of tries we get the automatic left-factorization of grammars that was mentioned in the beginning of the chapter.

4.2.2 Ambiguous grammars

The definition of the *parseFull* function reveals a slight inefficiency in the trie structure. In the middle case, when we are not at the end of the input, and have a trie with a result in it, we throw away the result and continue parsing with the "tail" of the trie. But if the grammar is very ambiguous, these intermediate results can make parsing slower, since we have to wander through every intermediate result in turn until we have reached a shift. For an ambiguous grammar we can have much more than one such garbage result in every node.

There is another way to implement tries which doesn't have these problems with ambiguous grammars. Instead of thinking of a node in the trie as a list of results with the finite map at the end, we will think of it as a pair of the list of results and the finite map.

```
data AmbTrie s α  =  [α] :&: Map s (AmbTrie s α)
```

It's straightforward to turn this trie into a premonad and to make it an instance of the Symbol class.

```
instance Ord s ⇒ PreMonad (AmbTrie s) where
  return a  =  [a] :&: emptyMap

instance Ord s ⇒ Symbol (AmbTrie s) s where
  sym s  =  [ ] :&: (s |→ return s)
```

The *zero* trie is a pair of the empty list and the empty map, and the choice consists of concatenating the results, together with merging the finite maps.

```
instance Ord s ⇒ Monoid (AmbTrie s) where
  zero  =  [ ] :&: emptyMap

  (as :&: pmap) <+> (bs :&: qmap)
        =  (as ++ bs) :&: mergeWith (<+>) pmap qmap
```

For the (≫=) combinator we have to collect all possible results of applying the continuation to the collected results, sum them together, and finally map the continuation on the sub-tries in the finite map.

```
instance Ord s ⇒ Monad (AmbTrie s) where
  (as :&: pmap) ≫= k  =  foldr (<+>)
                            ([ ] :&: mapMap (≫=k) pmap)
                            (map k as)
```

To parse a full sentence we simply go through the trie by looking up each symbol in turn and when the sentence is finished the lists of results is returned.

```
instance Ord s ⇒ Parser (AmbTrie s) s where
  parseFull (as :&: _)    [ ]        =  as
  parseFull (_ :&: pmap) (s : inp)   =  case pmap ? s of
                                          Just p   → parseFull p inp
                                          Nothing → [ ]
```

Exercise

Define the *parse* function on ambiguous tries and explain why this kind of trie is more suited for parsing full sentences. □

4.3 Memory efficient tries

A finite letter tree can only be used to represent a finite language. But context-free grammars are used to represent infinite languages, which means that the language $\{three, thirty, four, forty\}^*$ from the beginning of this chapter, will be calculated to an infinite trie, as in figure 4.2.

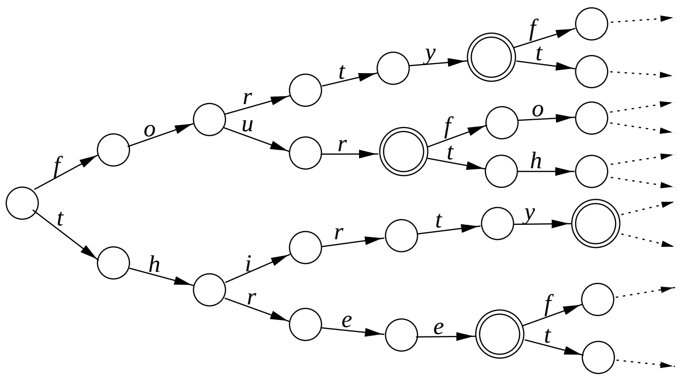


Figure 4.2: The trie of the $numbers_0$ parser.

This language is recognized by the $numbers_0$ recognizer, and the question arises how this will be stored in the memory of a lazy functional language, which shares the common structures in a program.

$$numbers_0 \quad = \quad many_0\ number$$

To start with, we need to know how the *number* parser is stored in memory. We will show the parser as a graph, with arrows to picture the finite maps, see figure 4.3

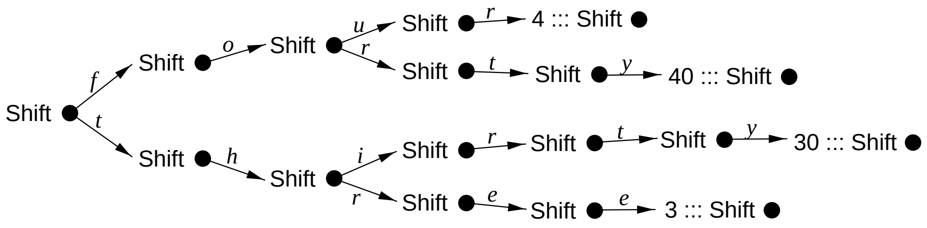


Figure 4.3: The memory structure for the *number* trie.

4.3.1 Lazy tries are finite automata

The sharing behaviour of a lazy functional language turns the tree structure of a trie into a directed graph, since two sub-tries can be shared. It is also possible to have cycles in the graph, if a trie contains itself as sub-trie.

Consider the simple ab_0 recognizer, defined as $many_0$ (*sym* `'a'` <+> *sym* `'b'`). When parsing this trie, Haskell will expand the definitions of the combinators and create a trie structure in memory. By doing the expansions ourselves we can informally reason about its space behaviour.[2]

$$
\begin{array}{lll}
ab_0 & \equiv & many_0\ (sym\ \texttt{'a'} <\!\!+\!\!> sym\ \texttt{'b'}) \\
 & \equiv & \textbf{let}\ ps = return\ () <\!\!+\!\!> ((sym\ \texttt{'a'} <\!\!+\!\!> sym\ \texttt{'b'}) \gg ps)\ \textbf{in}\ ps \\
 & \equiv & \textbf{let}\ ps = (() ::: zero) <\!\!+\!\!> ((sym\ \texttt{'a'} <\!\!+\!\!> sym\ \texttt{'b'}) \gg ps)\ \textbf{in}\ ps \\
 & \equiv & \textbf{let}\ ps = () ::: (zero <\!\!+\!\!> ((sym\ \texttt{'a'} <\!\!+\!\!> sym\ \texttt{'b'}) \gg ps))\ \textbf{in}\ ps \\
 & \equiv & \ldots \\
 & \equiv & \textbf{let}\ ps = () ::: (\mathsf{Shift}\ [\,(\texttt{'a'},\ \texttt{'a'} ::: zero), \\
 & & \qquad\qquad\qquad\qquad (\texttt{'b'},\ \texttt{'b'} ::: zero)\,] \gg ps)\ \textbf{in}\ ps \\
 & \equiv & \textbf{let}\ ps = () ::: \mathsf{Shift}\ [\,(\texttt{'a'},\ (\texttt{'a'} ::: zero) \gg ps), \\
 & & \qquad\qquad\qquad\qquad (\texttt{'b'},\ (\texttt{'b'} ::: zero) \gg ps)\,]\ \textbf{in}\ ps \\
 & \equiv & \ldots \\
 & \equiv & \textbf{let}\ ps = () ::: \mathsf{Shift}\ [\,(\texttt{'a'},\ ps),\ (\texttt{'b'},\ ps)\,]\ \textbf{in}\ ps
\end{array}
$$

Out informal reasoning therefore suggests that the ab_0 recognizer will be expanded to $() ::: \mathsf{Shift}\ [\,(\texttt{'a'},\ ab_0),\ (\texttt{'b'},\ ab_0)\,]$ while parsing, but then it is not necessary to expand it further, since the sub-tries will be shared with the main trie. The trie will thus be stored in memory as a graph with two cycles, see figure 4.4.

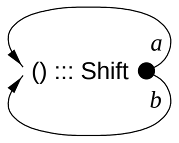


Figure 4.4: The memory structure for the ab_0 trie.

Observe that we have not proved anything, we have only tried to argue why a reasonable implementation of a lazy functional language should store the ab_0 recognizer as a cyclic graph. Informal tests performed for the Haskell implementations Hugs and GHC show that the reasoning really is correct. To prove the claim formally one could use e.g. the theory of weak space improvement by Gustavsson and Sands [12], but such a proof has yet to be done.

The $numbers_0$ recognizer is also stored in finite memory due to sharing. A similar reasoning as for the ab_0 recognizer reveals the memory behaviour in figure 4.5.

When parsing these tries, we do not have to recalculate the finite maps more than once because of sharing. This means that they behave just as deterministic finite automata.

[2]Observe that in this reasoning, we assume that the finite maps are stored as association lists. To simplify the reasoning, we also assume $zero <\!\!+\!\!> p \equiv p$ which is not quite the case, it is more like $zero <\!\!+\!\!> \mathsf{Shift}\ pmap \equiv \mathsf{Shift}\ pmap$. It is no real problem, since the eqivalence can be added as a case to the definition of $(<\!\!+\!\!>)$ by using the predicate *isEmptyMap*.

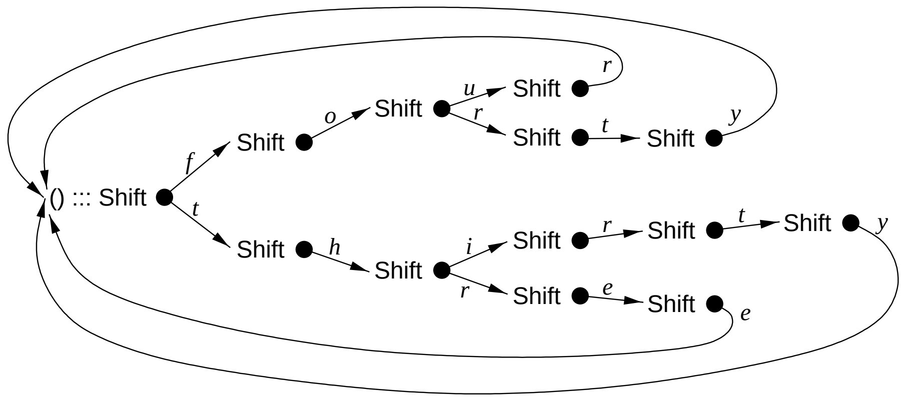


Figure 4.5: The memory structure for the *numbers*$_0$ trie.

Exercise

Manuel Chakravarty has described an efficient lexer using trie structures to represent finite automata [4]. Read his paper and discuss the similarities with our approach. How can we define his combinators using the combinators defined in this thesis?

□

4.3.2 Failure of sharing

Unfortunately the construction above only works when we have cycles created by the ($\gg$) constructor; i.e. when we want to throw away the results of the cycle. The *many* parser on the other hand returns a list of the results that are found on the way. This means that if we define the *ab* parser as *many* (*sym* `'a'` <+> *sym* `'b'`) we get a trie that takes up infinite memory.[3]

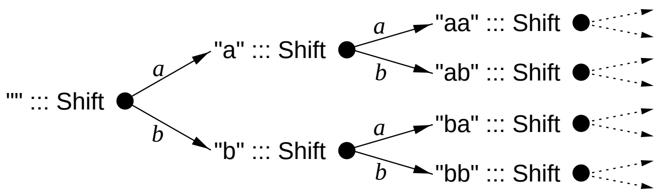


Figure 4.6: The memory structure for the *ab* trie.

It is impossible for any part to be shared with any other, since all results are different from each other. Our *numbers* trie will also be infinite of exactly the same reason.

[3]Lazy evaluation saves us here by only calculating the parts of a trie that is necessary, but it is anyhow a problem.

4.3.3 Extended tries

It is possible to solve this problem partly – not for the general monadic bind, but for the *fmap* function, and therefore also for the (<⋆>), (<:>) and *many* combinators which are definable in terms of the *fmap* function. We simply add *fmap* as a constructor to the trie structure.

```
data ExTrie s α  =  Shift (Map s (ExTrie s α))
                 |  α ::: ExTrie s α
                 |  ∀β. FMap (β → α) (ExTrie s β)
```

This needs the extension of existential quantification in types [9], further described in the introduction, section 1.2.2. We use the constructor FMap as the implementation of the *fmap* function.

```
instance Ord s ⇒ Functor (ExTrie s) where
  fmap  =  FMap
```

We now need a way to map a function to a trie, which we will call *unfold*. The *unfold* function is like the traditional *fmap*, but it is only applied one step into the trie structure.

```
unfold   ::  Ord s ⇒ (α → β) → ExTrie s α → ExTrie s β
unfold f (Shift pmap)  =  Shift (mapMap (FMap f) pmap)
unfold f (a ::: p)     =  f a ::: FMap f p
unfold f (FMap g p)    =  FMap (f · g) p
```

The *zero* remains the same as before, and for the (<+>) we have to add cases for the new constructor. The only thing we can do when we stumble upon a FMap is to unfold the function into the trie.

```
instance Ord s ⇒ Monoid (ExTrie s) where
  zero                          =  Shift emptyMap

  (a ::: p)    <+> q            =  a ::: (p <+> q)
  p            <+> (b ::: q)    =  b ::: (p <+> q)
  FMap f p     <+> q            =  unfold f p <+> q
  p            <+> FMap f q     =  p <+> unfold f q
  Shift pmap <+> Shift qmap     =  Shift (mergeWith (<+>) pmap qmap)
```

We do not have to make any changes to the PreMonad and Symbol classes.

```
instance Ord s ⇒ PreMonad (ExTrie s) where
  return a  =  a ::: zero

instance Ord s ⇒ Symbol (ExTrie s) s where
  sym s  =  Shift (s |→ return s)
```

To implement (≫=) we again need to add a case for the new constructor, and still the only thing we can do is to unfold the function into the trie.

```
instance Ord s ⇒ Monad (ExTrie s) where
  Shift pmap ≫= k   =  Shift (mapMap (≫=k) pmap)
  (a ::: p)  ≫= k   =  k a <+> (p ≫= k)
  FMap f p   ≫= k   =  unfold f p ≫= k
```

When we want to parse an extended trie we could do it the simple way by just adding an extra case for the FMap constructor and applying the function to all results.

```
instance Ord s ⇒ Parser (ExTrie s) s where
  parse (FMap f p)    inp       =  [(inp′, f a) | (inp′, a) ← parse p inp]
  parse (a ::: p)     inp       =  (inp, a) : parse p inp
  parse (Shift _)     [ ]       =  [ ]
  parse (Shift pmap)  (s : inp) =  case pmap ? s of
                                     Just p  → parse p inp
                                     Nothing → [ ]
```

But this will create a series of calls to *map* (for lists) while parsing, which will consume memory, and another possibility is to add a continuation which will be applied in the end instead.

```
instance Ord s ⇒ Parser (ExTrie s) s where
  parse p inp  =  parse′ p inp id
    where parse′ (FMap f p)   inp       k  =  parse′ p inp (k · f)
          parse′ (a ::: p)    inp       k  =  (inp, k a) : parse′ p inp k
          parse′ (Shift _)    [ ]       k  =  [ ]
          parse′ (Shift pmap) (s : inp) k  =  case pmap ? s of
                                                Just p  → parse′ p inp k
                                                Nothing → [ ]
```

Of course, this version will consume memory by building a big series of function compositions which will not be resolved until the very end, but it is still an improvement.

Exercise

Define the *parseFull* function for parsing full sentences. □

Now assume that the *many* combinator is defined as in chapter 2, via applications of *fmap*

```
many p      =  ps
  where ps  =  return [ ] <+> (p <:> ps)
            =  return [ ] <+> (fmap (:) p <⋆> ps)
            =  return [ ] <+> (fmap (:) p ≫= λf → fmap f ps)
```

With this definition we can derive the following memory behaviour of the *ab* parser, defined previously as *many* (*sym* `'a'` <+> *sym* `'b'`).[4]

$$\begin{aligned}
ab \quad &\equiv \quad \mathit{many}\ (\mathit{sym}\ \texttt{'a'} \mathbin{<\!\!+\!\!>} \mathit{sym}\ \texttt{'b'})\\
&\equiv \quad \mathbf{let}\ \mathit{ps} = \mathit{return}\ [\,] \mathbin{<\!\!+\!\!>} (\mathit{fmap}\ (:)\ (\mathit{sym}\ \texttt{'a'} \mathbin{<\!\!+\!\!>} \mathit{sym}\ \texttt{'b'}) \gg\!= \\
&\qquad\qquad\qquad\qquad\qquad \lambda f \to \mathit{fmap}\ f\ \mathit{ps})\\
&\qquad \mathbf{in}\ \mathit{ps}\\
&\equiv \quad \mathbf{let}\ \mathit{ps} = [\,] ::: (\mathsf{FMap}\ (:)\ (\mathsf{Shift}\ [\,(\texttt{'a'},\ \texttt{'a'} ::: \mathit{zero}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{'b'},\ \texttt{'b'} ::: \mathit{zero})\,]) \gg\!= \\
&\qquad\qquad\qquad \lambda f \to \mathsf{FMap}\ f\ \mathit{ps})\\
&\qquad \mathbf{in}\ \mathit{ps}\\
&\equiv \quad \mathbf{let}\ \mathit{ps} = [\,] ::: (\mathsf{Shift}\ [\,(\texttt{'a'},\ (\texttt{'a'}:) ::: \mathit{zero}),\\
&\qquad\qquad\qquad\qquad (\texttt{'b'},\ (\texttt{'b'}:) ::: \mathit{zero})\,] \gg\!= \\
&\qquad\qquad\qquad \lambda f \to \mathsf{FMap}\ f\ \mathit{ps})\\
&\qquad \mathbf{in}\ \mathit{ps}\\
&\equiv \quad \mathbf{let}\ \mathit{ps} = [\,] ::: \mathsf{Shift}\ [\,(\texttt{'a'},\ ((\texttt{'a'}:) ::: \mathit{zero}) \gg\!= \lambda f \to \mathsf{FMap}\ f\ \mathit{ps}),\\
&\qquad\qquad\qquad\qquad (\texttt{'b'},\ ((\texttt{'b'}:) ::: \mathit{zero}) \gg\!= \lambda f \to \mathsf{FMap}\ f\ \mathit{ps})\,]\\
&\qquad \mathbf{in}\ \mathit{ps}\\
&\equiv \quad \mathbf{let}\ \mathit{ps} = [\,] ::: \mathsf{Shift}\ [\,(\texttt{'a'},\ \mathsf{FMap}\ (\texttt{'a'}:)\ \mathit{ps}),\\
&\qquad\qquad\qquad\qquad (\texttt{'b'},\ \mathsf{FMap}\ (\texttt{'b'}:)\ \mathit{ps})\,]\\
&\qquad \mathbf{in}\ \mathit{ps}
\end{aligned}$$

In figure 4.7, we see how the *ab* trie is stored in memory by sharing its subtrees. The same, but more complicated, reasoning leads to a memory behaviour of the new version of the *numbers* trie as shown in figure 4.8

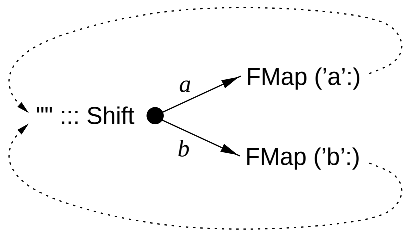


Figure 4.7: The memory structure for the new *ab* trie.

There are still many possibilities for creating infinite tries – one is just to merge a cyclic trie with itself. There is no way for the combinators to realize that the functions in the FMap constructors really are the same, so the tries will be unfolded to infinity. This means that even though the *ab* and *numbers* tries are finite, the *ab* <+> *ab* and *numbers* <+> *numbers* tries will be infinite.

[4]We make the same assumptions about association lists and *zero* <+> $p \equiv p$ as for the ab_0 recognizer above.

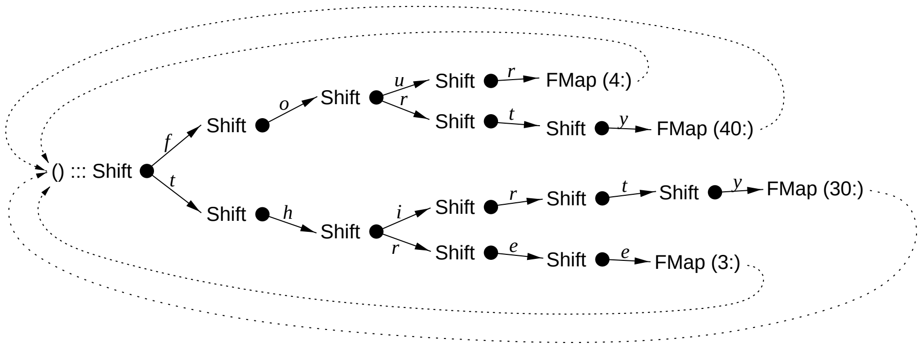


Figure 4.8: The memory structure for the the new *numbers* trie.

Exercise

Why is it not possible to add an extra constructor for (≫=) instead of *fmap*, as in the following definition?

$$\begin{array}{lcl} \textbf{data}\ \mathsf{MonadTrie}\ s\ \alpha & = & \mathsf{Shift}\ (\mathsf{Map}\ s\ (\mathsf{MonadTrie}\ s\ \alpha)) \\ & | & \alpha\ :::\ \mathsf{MonadTrie}\ s\ \alpha \\ & | & \forall\beta.\ \mathsf{MonadTrie}\ s\ \beta :\gg=: (\beta \to \mathsf{MonadTrie}\ s\ \alpha) \end{array}$$

□

4.3.4 Ambiguous grammars

We can do the same kind of *fmap* extension to the ambiguous trie from section 4.2.2, by adding a new constructor FMap to the data structure. This will increase the possibilities of sharing in the same way as for the ExTrie parser.

$$\begin{array}{lcl} \textbf{data}\ \mathsf{AmbExTrie}\ s\ \alpha & = & [\,\alpha\,] :\&: \mathsf{Map}\ s\ (\mathsf{AmbExTrie}\ s\ \alpha) \\ & | & \forall\beta.\ \mathsf{FMap}\ (\beta \to \alpha)\ (\mathsf{AmbExTrie}\ s\ \beta) \end{array}$$

Exercise

Give the definitions of the combinators and the *parseFull* function for this new trie structure.

□

4.4 Parsing to parsers

Doaitse Swierstra has in recent papers [39, 41] devoted some time to implement an efficient left-factorizing parser, and of course it is interesting to see where his work fits in this framework.

The main idea is to combine the standard backtracking parsers from chapter 3 with trie structures, hopefully getting the advantages of both approaches. The backtracking parsers are efficient on deterministic grammars, and the tries are efficient for ambiguities. The basic idea of Swierstra's parser is to use a standard parser for the deterministic parts of the grammar and a trie for the choices.

Swierstra's solution is to define a trie which contains *parsers*, not result values. When we come to a choice in the grammar, we parse the trie until we come to a deterministic part of the grammar. Then the trie will give us an efficient deterministic parser with which we can proceed with the parsing. Until we again come to a choice, when we parse the trie instead, and so on.

4.4.1 A trie of parsers

After removing the extra facilities for error reporting and error correction, which we don't cover in this thesis, and making the types more readable, we end up with a type very similar to our basic trie, but with one extra constructor. The basic idea of this trie structure is that it contains *parsers*, not values.

```
data ParserTrie s α  =  Shift (Map s (ParserTrie s α))
                     |  α ::: ParserTrie s α
                     |  Found α (ParserTrie s α)
```

The idea with the Found constructor is that if we reach it, we know how to proceed with the parsing in a deterministic way, and thus do not have to look any further in the trie. The second argument to Found is the trie to use when merging two tries. The Found constructor is really only used in the definition of the *sym* parser, since that parser is fully deterministic. The choice (<+>) removes all Found constructors since the result is potentially non-deterministic. Sequencing (<⋆>) keeps the structure of the trie, thus keeping the Found constructors, but does not introduce any more.

Since the choice is potentially non-deterministic, we simply remove all Found constructors. Otherwise the monoid instance is the same as for the standard trie.

```
instance Ord s ⇒ Monoid (ParserTrie s) where
  zero                               =  Shift emptyMap

  Found p ptrie <+> qtrie            =  ptrie <+> qtrie
  ptrie         <+> Found q qtrie    =  ptrie <+> qtrie
  (p ::: ptrie) <+> qtrie            =  p ::: (ptrie <+> qtrie)
  ptrie         <+> (q ::: qtrie)    =  q ::: (ptrie <+> qtrie)
  Shift ptries  <+> Shift qtries     =  Shift (mergeWith (<+>) ptries qtries)
```

It is possible to map a function on parsers onto the trie structure, which means we can make it an instance of the Functor class.

```
instance Ord s ⇒ Functor (ParserTrie s) where
  fmap f (Shift pmap)       =  Shift (mapMap (fmap f) pmap)
  fmap f (p ::: ptrie)      =  f p ::: fmap f ptrie
  fmap f (Found p ptrie)    =  Found (f p) (fmap f ptrie)
```

Finally we can parse the trie to get a list of possible parsers to use. As soon as we see a Found we have found a matching parser and do not have to look any further in the input. The other cases are just like the parsing function for the standard trie.

```
instance Ord s ⇒ Parser (ParserTrie s) s where
  parseFull (Found p _)  inp         =  [ p ]
  parseFull (p ::: ptrie)  inp       =  p : parseFull ptrie inp
  parseFull (Shift _)      [ ]       =  [ ]
  parseFull (Shift pmap) (s : inp)   =  case pmap ? s of
                                          Just ptrie → parseFull ptrie inp
                                          Nothing    → [ ]
```

4.4.2 Pairing a trie with a parser

To save unnecessary work we pair the trie with the underlying parser, which in turn is defined in terms of parts in the trie. The underlying parser will be a fast deterministic backtracking parser, such as the standard continuation parser from section 3.3.2, or the stack continuation parser from section 3.4. We will call this parser the real-parser in this section.

```
data PairTrie m s α  =  ParserTrie s (m α) :&: m α
```

We also have to define how to create a real-parser from a trie structure. To do that we parse the trie with the input to get some parsers which we can join together and use to parse the input. This sounds like duplicated work, and indeed in the worst case it can lead to duplicated work. But for real-world grammars, especially deterministic grammars, we will reach a Found constructor quite soon and do not have to look any further into the trie structure.

```
makeParser :: (Ord s, Monoid m, Lookahead m s) ⇒ ParserTrie s (m α) → m α
makeParser ptrie  =  lookahead (anyof · parseFull ptrie)
```

Here we must make use of a function *lookahead*, which creates a parser from a function on the input.

```
class Lookahead m s | m → s where
  lookahead  ::  ([ s ] → m α) → m α
```

The standard parser is an instance of the Lookahead class, as well as the continuation transformer and the stack continuation transformer.

```
instance Lookahead (Standard s) s where
  lookahead f   =   Std (λinp → let Std p = f inp in p inp)


instance Lookahead m s ⇒ Lookahead (ContTrans m) s where
  lookahead f   =   CT (λfut → lookahead (λinp → let CT p = f inp in p fut))
```

Exercise

Define the Lookahead instance for the stack continuation parser. □

The *zero* of the paired parser is the pair of the trie's zero and the real-parser's zero. To join two parsers, we calculate a new trie structure from the given tries, and create a new real-parser from the trie structure. Observe that we throw away the given real-parsers, because they do not correspond to the resulting trie structure.

```
instance (Ord s, Monoid m, Lookahead m s) ⇒ Monoid (PairTrie m s) where
  zero                              =   zero :&: zero

  (ptrie :&: _) <+> (qtrie :&: _)   =   pqtrie :&: makeParser pqtrie
    where pqtrie                    =   ptrie <+> qtrie
```

To return a result, we pair the underlying *return* real-parser with a trie that returns that parser.

```
instance (Ord s, PreMonad m) ⇒ PreMonad (PairTrie m s) where
  return a      =   (p ::: zero) :&: p
    where p     =   return a
```

The *sym* parser will be a pair of a trie structure and the natural real-parser. The trie structure will be a Found to say that we do not have to look any further ahead to know what parser we want to use. The rest of the trie structure is what will be used if we want to join this parser with another one sometime. This trie shifts the symbol to a trie that returns a *skip* real-parser.

```
instance (Ord s, Symbol m s) ⇒ Symbol (PairTrie m s) s where
  sym s            =   Found p ptrie :&: p
    where p        =   sym s
          ptrie    =   Shift (s |→ Found skip (skip ::: zero))
```

Recall that the *skip* parser doesn't look at the next input symbol at all, which makes it faster than the *sym* parser since it doesn't have to do any comparisions.

To sequence two parsers, we create a brand new trie structure by adding the second trie to the ends of the first one. And whenever we have a Found parser in the first trie, we sequence that parser with the second's real-parser.

```
instance (Ord s, Monoid m, Sequence m, Lookahead m s)
                                   ⇒   Sequence (PairTrie m s) where
  (ptrie :&: _) <⋆> ~(qtrie :&: q)  =   pqtrie :&: makeParser pqtrie
    where pqtrie                    =   mapPQ ptrie
          mapPQ (Shift pmap′)       =   Shift (mapMap mapPQ pmap′)
          mapPQ (p′ ::: ptrie′)     =   mapPQ ptrie′
                                  <+>   fmap (p′<⋆>) qtrie
          mapPQ (Found p′ ptrie′)   =   Found (p′ <⋆> q) (mapPQ ptrie′)
```

Observe that it is important that we pattern-match lazily on the second argument (which is denoted in Haskell by the preceding ∼), to stop us from falling into a right-recursive trap.

Finally, to parse with a given parser, we can use the real-parser that has been created from the trie. Observe that we cannot define the *parse* function, since the *makeParser* function looks at the whole input sequence.

```
instance (Ord s, Parser m s) ⇒ Parser (PairTrie m s) s where
  parseFull (_ :&: p)  =  parseFull p
```

The pairing of a trie and a continuation parser makes it difficult to understand how this parser really works. The simple explanation is that we use the continuation parser where it is efficient – for deterministic parsing – and the trie structure where it is appropriate – for the non-deterministic choices in the grammar.

If we try to parse the string `"threeforty"` using the *numbers* parser, the parser will behave something like the following. The parser first walks through the trie for the initial `"thr"`, then it can use the continuation parser for the following `"ree"` until it comes to a choice again where it must walk through the trie for the next `"for"`, and then finally it can use the continuation parser for the last `"ty"`.

4.5 Discussion

Trie structures and LL(k)-grammars An LL(k) grammar is a grammar which can be parsed in linear time by a recursive-descent parser with k symbols of lookahead. Trie structures are deterministic, which means that a trie can always be parsed in linear time through recursive-descent parsing. Thus a trie structure can be said to implement an LL(∞) parser. But this is not entirely true – since the trie structures are computed during parsing, the parsing will not be linear in general.

Regular expressions and finite automata That tries can be used to compile regular expressions to finite automata has already been noticed by Chakravarty [4]. Here we go a step further to include polymorphic parse results, by defining the ExTrie constructor.

Chapter 5

Chart Parsing

One particular algorithm, or rather a family of algorithms, common for natural language grammars, is chart parsing [20, 46]. The algorithm is descended from the algorithms of Earley [8] and Cocke, Kasami and Younger [19, 47]. In chart parsing there are different strategies – you can parse top-down or bottom-up, and combinations, with or without filtering. In this chapter we implement unfiltered bottom-up chart parsing à la Kilbury [21].

Chart parsing is a family of parsing algorithms, well suited for linguistic applications. It is a generalization of the algorithms of Earley [8] and Cocke, Younger and Kasami [19, 47], of which the latter is described in more detail in chapter 7. Chart parsing has been described by Kay [20] and Wirén [46] among others, and in the last years it has been even more generalized to the ideas of deductive parsing [37] parsing schemata [38].

5.1 Edges and the chart

A chart is a set of items called edges. Each edge is of the form $\langle i, j : A/\beta \rangle$, where $0 \leqslant i \leqslant j$ are integers, A is a category and β is a sequence of categories. If β is empty the edge is called passive, and we write it as $\langle i, j : A \rangle$; otherwise it is called active. For an input of n words, we create $n + 1$ nodes labelled $0 \ldots n$. The i:th word in the input will then span the nodes $i - 1$ to i.

The intended meaning of a passive edge $\langle i, j : A \rangle$ is that we have found the category A spanning the nodes i and j. The meaning of an active edge $\langle i, j : A/\beta \rangle$ is that if we can find the categories β between j and some k, then we will know that A spans between i and k.

The key idea of chart parsing is that we start with an empty chart, to which we add new edges by applying some inference rules. We write the rules in a natural deduction style, where the following rule means that if the edges $e_1 \dots e_k$ are in the chart, and the property ϕ holds, add the edge e to the chart.

$$\frac{e_1 \ \dots \ e_k}{e} \ \phi$$

The properties are either on the form $A \to \beta$, which means that that particular production is in the grammar; or on the form $\alpha \Rightarrow^* \epsilon$, to say that the sequence α is nullable, i.e. it only consits of empty categories. We write t_i for the i:th token in the input sequence.

5.2 Kilbury bottom-up chart parsing

There are different strategies for chart parsing, which are reflected in the inference rules. In the bottom-up strategy, we start by adding the input tokens as passive edges, and then build the chart upwards. In this first version we assume that we have no empty productions in the grammar, i.e. rules of the kind $A \to \epsilon$. This has the effect that there will never be any edges on the form $\langle i, i : A/\beta \rangle$ in the chart, which means that the corresponding graph will be acyclic. This in turn makes it possible to implement the algorithm elegantly in Haskell. In section 5.5 we will add specific inference rules for empty productions, to keep the chart acyclic.

Scan All the categories for the k:th input token t_k are added as passive edges spanning the nodes $k-1$ and k. As mentioned in section 2.1, we assume the terminals only appear in unit productions, which means that this is the only necessary inference rule dealing with terminals.

$$\frac{}{\langle k{-}1, k : A \rangle} \ A \to t_k$$

Predict If we have found a passive edge for the category A, spanning the nodes j and k, and there is a production in the grammar that looks for an A, we add that production as an edge. And since we have found the category A in the right-hand side, we only have the categories after A left to look for.

$$\frac{\langle j, k : A \rangle}{\langle j, k : B/\beta \rangle} \ B \to A\beta$$

This particular variant of bottom-up parsing is called Kilbury parsing, first described in [21]. The difference to the traditional bottom-up strategy is that we do not add $\langle j, j : B/A\beta \rangle$ as an edge, since we anyhow will end up in the edge above. Apart from saving some extra work, it will also help us keeping the chart acyclic.

Combine If we have an active edge looking for a A at the j:th node, and there is a passive edge labelled A spanning j and k, we can move the active edge forward to the k:th node.

$$\frac{\langle i,j : B/A\beta\rangle \quad \langle j,k : A\rangle}{\langle i,k : B/\beta\rangle}$$

The parsing succeeds if there exists a passive edge for the starting category, spanning the whole chart; i.e. if $\langle 0,n : S\rangle$ is in the chart, where n is the number of input tokens.

Depending on the parsing strategy the inference rules might look different, and there can be even more rules. But one rule always exist in all chart parsing algorithms, the Combine rule, also called "the fundamental rule of chart parsing".

5.2.1 The chart as a directed graph

A nice way to visualize a chart is as a directed graph. The graph is almost acyclic, because we can have edges going from one node to itself, but never backwards to a previous node. As an example, suppose that we have the following fragment of an English grammar.

S	$\longrightarrow$	NP VP							
VP	$\longrightarrow$	Verb	|	Verb NP					
NP	$\longrightarrow$	Noun	|	Det Noun	|	NP PP			
PP	$\longrightarrow$	Prep NP							
Verb	$\longrightarrow$	*flies*	|	*like*	|	...			
Noun	$\longrightarrow$	*flies*	|	*time*	|	*arrow*	|	...	
Det	$\longrightarrow$	*an*	|	...					
Prep	$\longrightarrow$	*like*	|	...					

In figure 5.1 we see the chart after the first four words in the sentence "time flies like an arrow". The dotted arrows denote the edges that will be created when the fifth word has been incorporated. Each arrow really symbolizes a set of edges, which are written above and below the edge. E.g. there are seven edges between node 1 and 2, of which three are active and four passive.

5.3 Kilbury parsing in Haskell

We will implement chart parsing in an incremental way, by starting with the first input token and then applying all the inference rules. Then we add the

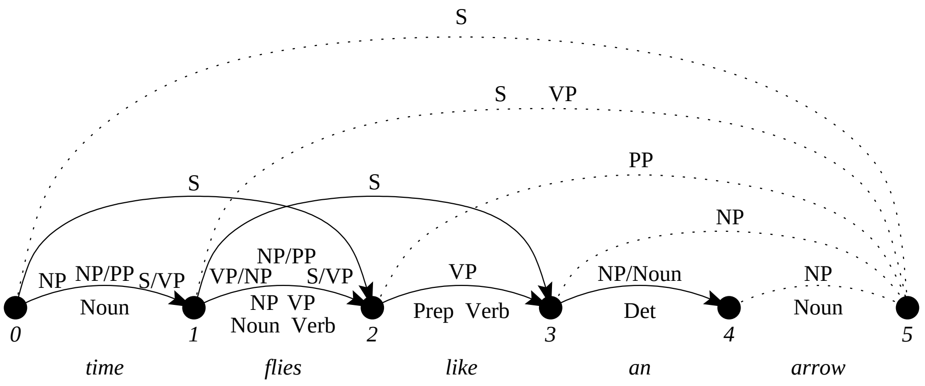


Figure 5.1: The chart after the first four words have been incorporated.

second token and apply the inference rules again. And so on until we have added the last token, when we will have a final chart.

This strategy makes it possible to represent the chart as a list of states, each state being all the edges ending in a particular node. The states will be called Earley states, since this is how to represent the parsing state in Earley's original parsing algorithm, which in turn can be seen as an implementation of top-down chart parsing. So, a chart will be a list of sets of edges.

```
type Chart c  =  [State c]
type State c  =  Set (Edge c)
```

The edge $\langle j, k : A/\alpha\rangle$ is a 4-tuple of the two nodes j and k, the category A and the list of categories α. But the ending node k is not necessary to remember, since it is implicit in the position of the state in the chart list.

```
type Edge c  =  (Int, c, [c])
```

The main function builds a chart from a given grammar and the input sequence.

```
buildChart  ::  Ord c ⇒ Grammar c t → [t] → Chart c
buildChart (_, _, terminal, productions) input  =  finalChart
  where finalChart    =  map buildState initialChart
        initialChart  =  ...
        buildState    =  ...
```

For the sake of presentation we will define the rest of the functions on the top-level, but in reality they need to be in the scope of the grammar and the input sequence.

The initial chart consists of the results of the Scan inference rule applied to the input tokens. Each Earley state k (except from the empty 0:th state) will consist of all the edges $\langle k{-}1, k : A\rangle$ such that $A \to t_k$, where t_k is the k:th input token.[1]

```
initialChart  ::  Chart c
initialChart  =   emptySet : map initialState (zip [0 ..] input)
  where initialState (j, sym)
              =   ordSet [(j, cat, [ ]) | cat ← elems (terminal sym)]
```

Now we have to argue that we are allowed to use the *ordSet* function, i.e. that the list is ordered. But since the *elems* function returns an ordered list, and the list comprehension doesn't change the order, the argument to *ordSet* will be ordered.

Both the Predict and the Combine inference rule only apply to passive edges, which means that an active edge will not lead to any new edges being added to the chart. The reason why the Combine rule only applies to passive edges is that the active edge $\langle i, j : A/B\beta\rangle$ ends in the j:th node, and since we have no cycles in the graph, $j < k$, where k is the Earley state to which the new edge $\langle i, k : A/\beta\rangle$ will be added.

```
buildState   ::   State c → State c
buildState   =    limit more
  where more (j, a, [ ])
             =    ordSet [(j, b, bs) |
                          (b, a′ : bs) ← elems productions, a == a′]
          <++>    ordSet [(i, b, bs) |
                          (i, b, a′ : bs) ← elems (finalChart !! j), a == a′]
        more (j, b, a : bs)
             =    emptySet
```

The first list comprehension for the *more* function corresponds to the Predict inference rule, and the second to the Combine rule. The Predict rule only has to find the productions in the grammar that are looking for the found category a, and the Combine rule searches in the previous j:th Earley state for the active productions looking for an a.

Observe that *buildState* makes use of *finalChart*, which in turn is built by calling the function *buildState*. This is permitted because *buildState* only looks up previously built states in *finalChart*, and we use lazy evaluation. This technique helps us to write a loop in an imperative algorithm in a clean functional way.

We finally have to prove that the arguments to the two applications of the *ordSet* function are ordered lists. But this is true since the variables a and j are fixed and the order between the other variables is retained in the comprehensions.

[1]We do not write $k - 1$ in the code, since it is already accounted for in the *zip* [0 . .] application.

5.4 Building the parse trees

To build the parse trees, we only need the passive edges from the chart. Also, things become simpler if we transform the list of Earley states into a big collection of all the passive edges.[2]

```
type Passive c         =  (Int, Int, c)

passiveEdges           :: Chart c → [Passive c]
passiveEdges chart     =  [(i, j, cat) |
                            (j, state) ← zip [0 ..] chart,
                            (i, cat, [ ]) ← elems state]
```

Assuming that we already have built a collection of passive edges, we can pair each edge with the parse trees corresponding to that edge.

```
buildTrees    :: Grammar c t → [t] → [Passive c]
                         →  [(Passive c, [ParseTree c t])]
buildTrees (_, _, terminal, productions) input passiveChart = edgeTrees
  where edgeTrees    =  [(edge, treesFor edge) | edge ← passiveChart]
        treesFor     =  . . .
```

To construct the parse trees of an edge $\langle i, j : A\rangle$, we find each matching production $A \to \alpha$ and get its right-hand side as a list of categories α. Then the *children* function tries to find a path for α from i to j in the chart, while collecting the parse trees for all the visited edges. Since there can be many different paths for α between i and j, we can get many sequences of trees as the result of the *children* function, and every one of these sequences are used to form a parse tree for our given edge.

There is also the possibility that the edge was created by the Scan inference rule, in which case we just add the parse tree for the i:th input token.

```
treesFor    :: Passive c → [ParseTree c t]
treesFor (i, j, cat)   =   [cat :^ trees |
                             (cat′, rhs) ← elems productions,
                             cat == cat′,
                             trees ← children rhs i j]
                       ++  [cat :^ [Leaf sym] |
                             i == j − 1,
                             let sym = input !! i,
                             cat ‘elemSet‘ terminal sym]
```

To collect the trees of the children, we make use of lazy evaluation and simply lookup the trees in the list *edgeTrees* of edges and trees we are in the process of

[2]The collection will not necessarily be ordered, so we can not give it the type Set Edge, but there will be no duplicated edges since it is calculated from a set.

computing.

```
children   ::   [c] → Int → Int → [[ParseTree c t]]
children []       i k  =  [[] | i == k]
children (c : cs) i k  =  [tree : rest |
                            i ⩽ k,
                            ((i', j, c'), trees) ← edgeTrees,
                            i == i', c == c',
                            rest ← children cs j k,
                            tree ← trees]
```

Observe that it is important to call the functions in the correct order – we must collect the tail of the trees we are constructing before we take out one tree at a time. Otherwise we get into an infinite loop because we try to look into the trees before we have constructed them.

Finally we can collect the parse trees that correspond to an edge for the starting category, spanning the whole input.

```
parse   ::   Ord c ⇒ Grammar c t → [t] → [ParseTree c t]
parse grammar@(_, start, _, _) input
    = case lookup (0, length input, start) edgeTrees of
          Just trees      →  trees
          Nothing         →  []
      where edgeTrees     =  buildTrees grammar input passiveChart
            passiveChart  =  passiveEdges finalChart
            finalChart    =  buildChart grammar input
```

5.5 Adding empty productions

To be able to parse a grammar containing empty productions, i.e. productions on the form $A \rightarrow \epsilon$, we have to change the inference rules slightly.

Scan This inference rule remains the same as before.

$$\frac{}{\langle k{-}1, k : A\rangle}\; A \rightarrow t_k$$

Predict When we allow empty productions, a production can start with a sequence of categories which are all empty. So, when we want to predict a new possible edge, we skip the initial empty categories and go directly to the first non-empty category.

$$\frac{\langle j, k : A\rangle}{\langle j, k : B/\beta\rangle}\; B \rightarrow \alpha A \beta,\ \alpha \Rightarrow^* \epsilon$$

Combine The traditional inference rule remains the same, but we also have to add another Combine rule. When we have an edge that looks for a category A which is empty, i.e. can be rewritten to ϵ, we can simply skip that category.

$$\frac{\langle i, j : A/B\beta\rangle \quad \langle j, k : B\rangle}{\langle i, k : A/\beta\rangle} \qquad\qquad \frac{\langle j, k : B/A\beta\rangle}{\langle j, k : B/\beta\rangle}\ A \Rightarrow^{*} \epsilon$$

This final algorithm is similar to the algorithm of Graham, Harrison and Ruzzo [11], with only minor deviations.

5.5.1 Adding empty productions to the Haskell version

Since we have added a new Combine rule and changed the Predict rule, our Haskell implementation should be changed. This means that the *buildState* function that builds an Earley state from the initial edges needs some fixing.

```
buildState    ::  State c → State c
buildState    =   limit more
  where more (j, a, [ ])
              =   makeSet [(j, b, bs) |
                               (b, abs) ← elems productions,
                               (a′ : bs) ← removeNullable abs, a == a′ ]
          <++>    ordSet [(i, b, bs) |
                               (i, b, a′ : bs) ← elems (finalChart !! j), a == a′ ]
        more (j, b, a : bs)
              =   ordSet [(j, b, bs) |
                               a ‘elemSet‘ empties grammar ]
```

The changes to the previous definitions are two:

- In the list comprehension corresponding to the Predict inference rule, we use a function *removeNullable* which removes possible empty categories from the beginning of a list of categories, i.e. it removes all possible initial sequences $\alpha \Rightarrow^{*} \epsilon$ from a given sequence.

  ```
  removeNullable  ::  [c] → [[c]]
  removeNullable  [ ]                         =  [ ]
  removeNullable cats@(cat : cats′)
       | cat ‘elemSet‘ empties grammar        =  cats : removeNullable cats′
       | otherwise                            =  [cats]
  ```

  Observe that this function does not necessarily return an ordered list, which makes it necessary to use the *makeSet* function instead of *ordSet* when defining *buildState*.

- We have added a list comprehension for the case of an active edge, which corresponds to the extra Combine inference rule.

5.5.2 Adding the parse trees

Since the cyclic passive edges corresponding to an empty production rule are not stored in the chart, we have to consider them while building the trees. And the only thing we have to do is to add all possible empty edges to the collection of passive edges, which is done by simply enumerating all possible nodes and empty categories.

```
passiveEdges        ::  Chart c → [Passive c]
passiveEdges chart   =  [(i, j, cat) |
                         (j, state) ← zip [0 ..] chart,
                         (i, cat, [ ]) ← elems state]
                    ++  [(i, i, cat) |
                          i ← [0 .. length input],
                          cat ← elems (empties grammar)]
```

5.6 Improving efficiency

There are some ways to improve the efficiency of our implementation. Since we are building the chart incrementally – building one Earley state fully before starting to build the next – we can transform the previous states into more efficient data structures when they are finished.

5.6.1 Analyzing the grammar

While building the chart and the parse trees we make heavy use of the grammar, and the first thing one can try is to make the lookup in the grammar more efficient.

In the Predict inference rule we search for a production whose leftmost category matches a certain one. Also, it skips empty categories in the production. To make this more efficient, one can transform the grammar into a balanced binary search tree in which categories can be looked up to immediately get the matching right-hand sides with the initial empty categories skipped. This turns the lookup of productions logarithmic in the size of the grammar instead of linear.

While building parse trees we also lookup categories in the grammar, but this time it is the main category of a production we are looking for, which means that we need another binary search tree to make this lookup more efficient.

Exercise

Implement these two finite maps as balanced binary search trees. Call the first one *leftcornerMap* and the second *grammarMap*. Change the implementations of the functions *buildState* and *treesFor* to use the finite maps instead of the set *productions* directly.

□

5.6.2 Efficient lookup in the chart

When we are building a new Earley state, we need to look up edges in the previous states. Here we can make two improvements.

First, we can turn the chart into an array indexed over integers, instead of a list. This will make it constant time to find a certain state in the chart. Second, we can turn each state into a finite map, since we are looking for a certain category in the state.

Exercise

Implement these two improvements on the chart, and change the *buildState* function correspondingly.

□

5.6.3 Efficient lookup for parse trees

While building the parse trees, the *children* function searches for the parse trees of a certain edge in the set *edgeTrees* of edges and trees. This can be improved upon by transforming the set into a finite map, where we can lookup edges more efficient.

Exercise

Implement this improvement on the *edgeTrees* set, and change the *children* function correspondingly. The lookup in the *parseResult* can also use this improvement.

□

5.7 Discussion

Space complexity An edge in the k:th Earley state ends in node k and can start in any node $j < k$. It can be derived from any of the productions in G, and up to δ categories can have been removed from the right-hand side of its production, where δ is the length of the longest production in G. This means that there will be $O(n|G|\delta)$ edges in the k:th Earley state, since $k = O(n)$. Since there are n states altogether, we will have $O(n^2|G|\delta)$ edges in the final chart.

Time complexity The Earley states are defined in terms of the *limit* function, which has a worst-case complexity of $O(m^2)$, where m is the size of the final Earley state. See section 2.2.1 for a discussion of the *limit* function. This gives us that the time to build one Earley state will be $O(n^2|G|^2\delta^2)$. And since we have n states in total, the worst-case complexity is $O(n^3|G|^2\delta^2)$. This is the standard complexity for imperative algorithms.

This result is not entirely true though, since the *more* function in *buildState* has to do some extra work to calculate the edges to be added. In the version in section 5.3 there are two list comprehensions – one going through all productions and one going through all edges in a previous state. The first comprehension has a complexity of $O(|G|)$ and the second a complexity of $O(n|G|\delta)$, which then have to be multiplied with the result above.

The improvements suggested in section 5.6 will only add an extra overhead of $O(\log |N|)$, for the logarithmic lookup time in binary trees, where $|N|$ is the number of non-terminals in the grammar. Thus the final worst-case complexity for chart parsing with the suggested improvements will be $O(n^3|G|^2 \log |N|\delta^2)$.

Building the parse trees The space and time complexity for building the parse trees is of course exponential, since there can be an exponential number of trees.

Lazy evaluation Lazy evaluation is used in two places; *i*) when building the final chart as a list of Earley states, where we refer back to the earlier states while building a new state; and *ii*) when calculating the parse trees, where we refer to the parse trees of an edge, sometimes even before the trees of that edge are calculated. Without laziness the code would be much clumsier and less declarative, making it harder to read and understand.

Also, laziness will only calculate the parse trees that are used in the final parse result. For our example grammar, this means that the parse tree of the sentence (S) "flies like an arrow" will never be calculated, only the parse tree of the corresponding verb phrase (VP).

Chapter 6

Generalized LR Parsing

In this chapter we define several versions of generalized LR parsing. The original deterministic LR algorithm due to Knuth [22] was extended by Lang [24] by using the LR table to process general context-free grammars. We start with a simple bottom-up approach and improve it step-wise until we reach an approximation of Tomita's efficient LR algorithm [42]. This is accomplished without using the complicated graph-structured stack used in the Tomita algorithm. The approximation comes from the fact that in Haskell we cannot know whether two structures are shared or not. Unfortunately this approximation does not have polynomial time complexity.

The LR parsing algorithm is one of the ancient ones, dating back to the mid 60's and Donald Knuth [22]. Most of the work in LR parsing has been done on deterministic LR grammars, but the ideas can be used to parse general context-free grammars. This was noted first by Lang in the 70's [24], but his algorithm was in the worst case exponential in the length of the input. It was not until the mid 80's when Tomita published his efficient parsing algorithm [42] that the ideas became useful for e.g. natural language parsing.

Since the term "LR parsing" is often used to describe deterministic parsing of LR grammars, we will follow the terminology in [38] and use the term "generalized LR parsing" for the algorithms in this chapter.

The special thing with LR parsing is that the grammar is analyzed and compiled into an "LR table" before the real parsing begins. While parsing we will only make use of the LR table, and not refer to the original grammar at all.

6.1 The LR table

Before all parsing begins, the grammar is analyzed and compiled into a finite automaton, with some extra facilities to handle the recursion in the grammar.

Here we will briefly describe the result of the compilation, and in section 6.5 later we will show an example of how the grammar can be analyzed.

All the functions described in this section, together with the type of LR states, will be defined in that later section. Observe that they all depend on the grammar, so in reality they will be defined as local functions inside the main *parse* function.

6.1.1 The grammar as a finite automaton

The grammar is compiled into a finite automaton over the categories of the grammar. The states of the automaton are called *LR states*, of which there is a starting state and a set of final states. We implement the final state set as a predicate *accept*.

```
type LRState c  =  ...
startState  ::  LRState c
accept      ::  LRState c → Bool
```

The transition function is called *shift*, and it takes an old LR state and a category, and returns a collection of new states.[1] The collection might be empty, in which case the shift is not possible.

```
shift  ::  LRState c → c → [LRState c]
```

6.1.2 The recursive parts of the grammar

A finite automaton can only recognize regular languages, but a context-free grammar can do more than that. The difference is that a context-free grammar is recursive, and we need a way to handle this to be able to use the automaton.

While traversing the automaton, the LR parser remembers the states it has visited in a stack of states. In some of the states it is possible to, instead of shifting to a new state, replace some of the top states of the stack with a new one. This is determined by the *reduce* function, which returns a collection of categories and integers, depending on the current state.[2]

```
reduce  ::  LRState c → [(c, Int)]
```

The integer determines how many states are to be popped off the stack, and then we use apply the *shift* function to the new current state and the returned category, to get the next state to be pushed onto the stack.

[1]In reality the automaton is deterministic, and it will never be possible to shift to more than one state. This means that we could use Maybe (LRState c) as the result type, but the code becomes simpler if we stick to lists.

[2]This particular definition of *reduce* gives rise to a LR(0) recognizer, where we do not care about the next input symbol when determining whether to reduce or not.

The *reduce* function applies when we have reached the end of a context-free production, say $A \to \beta$. The returned category will be A, and the integer represents the number of symbols on the right-hand side, i.e. the length of β. When we pop that number of states off the stack, we come to the state where we were before we started to recognize β; and *shift* tells us where to go now that we have recognized A.

6.1.3 A grammar of expressions

Consider the following grammar of simple mathematical expressions.

Sent	$\longrightarrow$	Expr				
Expr	$\longrightarrow$	Digit	|	Expr	Oper	Expr
Digit	$\longrightarrow$	`"5"`				
Oper	$\longrightarrow$	`"-"`				

The grammar is highly ambiguous, since the operator can associate both to the left and to the right. So the expression `"5-5-5-5"` can mean any of the following: `"((5-5)-5)-5"`, `"(5-(5-5))-5"`, `"(5-5)-(5-5)"`, `"5-((5-5)-5)"` and `"5-(5-(5-5))"`.

Figure 6.1 shows the automaton for this grammar. The reduce actions are shown at the states where they act. The starting state is 1 and there is one final state, number 2. In section 6.5 later, the automaton is explained more thoroughly.

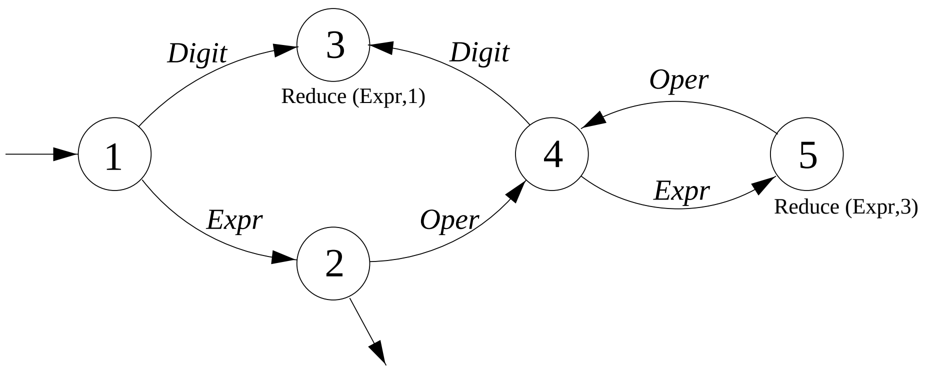


Figure 6.1: The LR automaton for the expression grammar.

6.2 LR parsing without parse results

In this section we describe three different versions of LR parsing, which we call depth-first, breadth-first and Tomita. They are given in order of simplicity, to be able to explain the efficient Tomita parser.

The parsers given in this section do not calculate any parse results. In the next section we will add parse trees to the implementations.

The implementations all define a function *processX* (where X depends on the version), taking a stack (or a collection of stacks), and a list of categories as arguments, and return a collection of stacks. The list of categories is the input to be recognized, and the result will be all final parse stacks, whose top state is accepted by the LR automaton.

6.2.1 A simple parse stack

Before we can describe a simple implementation of LR parsing, we must decide on how the parse stack should look like. The stack will have the usual three operations, *push*, *top* and *pop*. The *pop* operation will be a bit more general than usual – it will take a number as argument, and pop that many elements off the top of the stack.

For the first two algorithms we can simply say that a parse stack is a list of LR states.

```
type Stack c  =  [LRState c]
```

The current state will always be the top of the stack, which can be retrieved using the *top* operation.

Stack operations The stack operations are already defined in the Haskell prelude as the list constructor (:), and the functions *head* and *drop* respectively.

```
push  ::  LRState c → Stack c → Stack c
push  =   (:)

top   ::  Stack c → LRState c
top   =   head

pop   ::  Int → Stack c → Stack c
pop   =   drop
```

Reducing and shifting stacks Before we start defining the processing functions, we introduce two auxiliary functions $reduce_S$ and $shift_S$, which work on stacks, not states. $reduce_S$ reduces a stack any number of times, returning a collection of stacks. Firstly, the stack need not be reduced at all, so we return the stack itself. Secondly, we try to reduce the stack once by reducing the top state, popping a number of elements off the stack and pushing a new shifted state onto the popped stack. This gives us a number of new stacks, which in turn can be reduced any number of times.

```
reduce_S          ::  Stack c → [Stack c]
reduce_S stack    =   stack : concat [reduce_S (push state stack') |
                                 (cat, n) ← reduce (top stack),
                                 let stack' = pop n stack,
                                 state ← shift (top stack') cat]
```

shift$_S$ shifts a stack on a given terminal symbol. This can give rise to any number of resulting stacks, since there can be many categories matching the terminal. The new shifted state is pushed onto the stack, for each category.

```
shift_S              ::  Stack c → t → [Stack c]
shift_S stack sym    =   [push state stack |
                          cat ← terminal sym,
                          state ← shift (top stack) cat]
```

6.2.2 Depth-first recognition

The depth-first version takes a parse stack and the input as arguments. The first thing to do is to reduce the stack any number of times, and try to shift on the input for each of the reduced stacks.

```
processDF               ::  Stack c → [t] → [Stack c]
processDF stack input   =   concat [shiftDF stack' input |
                                    stack' ← reduce_S stack]
```

If we have reached the end of the input we return the given stack, provided that the top state is a final state. Otherwise we shift on the next input symbol, pushing the new state on the stack. Then we continue processing the rest of the input.

```
shiftDF                       ::  Stack c → [t] → [Stack c]
shiftDF stack [ ]             =   [stack | accept (top stack)]
shiftDF stack (sym : input)   =   concat [processDF stack' input |
                                          stack' ← shift_S stack sym]
```

Often one uses LR parsers only for deterministic LR grammars, in which case there will only be one single choice every time. But we want to parse general context-free grammars, and therefore we simply concatenate all possible results from the shifts and the reduces. This gives a behaviour similar to the depth-first search strategy of the programming language Prolog – we try to follow a path as far as possible and then try the next one.

Exercise

Rewrite the list comprehensions in the Haskell **do**-notation, and use monoid operations instead of list concatenation, to make the result type of stacks abstract.

```
reduce_S    :: (Monoid m, Monad m) ⇒ Stack c → m (Stack c)
shift_S     :: (Monoid m, Monad m) ⇒ Stack c → t → m (Stack c)
processDF   :: (Monoid m, Monad m) ⇒ Stack c → [t] → m (Stack c)
shiftDF     :: (Monoid m, Monad m) ⇒ Stack c → [t] → m (Stack c)
```

□

6.2.3 Breadth-first recognition

The depth-first version works fine on deterministic and almost-deterministic grammars, but not as well as we want on ambiguous grammars such as natural language grammars. The problem is that on every ambiguity, we will create new parse stacks, thus leading to an exponential number of stacks in the worst case.

The solution is to merge the stacks while parsing, to reduce the exponential explosion. But to be able to do this we have to parse the input breadth-first instead of depth-first. I.e. we have to process all possible stacks at once, and shift on them simultaneously. The new processing function will therefore work on a list of stacks. It reduces all stacks at once and shifts on them simultaneously.

```
processBF :: [Stack c] → [t] → [Stack c]
processBF stacks input = shiftBF stacks' input
  where stacks'        = concat [reduce_S stack | stack ← stacks]
```

If there is no input left, *shiftBF* filters out the stacks whose top state is accepted. If there are input symbols left, it shifts on each of the stacks. Then it continues processing the rest of the input.

```
shiftBF :: [Stack c] → [t] → [Stack c]
shiftBF stacks [ ] = [stack | stack ← stacks, accept (top stack)]
shiftBF stacks (sym : input)
                   = processBF stacks' input
  where stacks'    = concat [shift_S stack sym | stack ← stacks]
```

Exercise

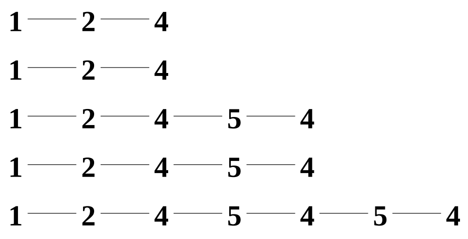
Do the same as in the previous exercise; rewrite the list comprehensions in **do**-notation, and use monoid operations instead of list concatenation, to abstract over the list of stacks.

```
processBF :: (Monoid m, Monad m) ⇒ m (Stack c) → [t] → m (Stack c)
shiftBF   :: (Monoid m, Monad m) ⇒ m (Stack c) → [t] → m (Stack c)
```

□

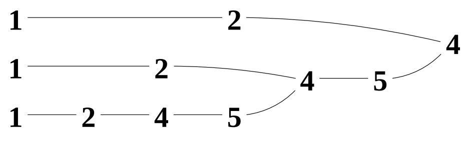
6.2.4 Tomita parsing

The breadth-first version is approximately as fast as the depth-first version.[3] They both work on the same number of stacks, only with different search strategies. What we want to do now is to reduce the number of stacks to work on. Consider the ambiguous grammar of expressions described in section 6.1.3. After having processed the first six symbols of the input `"5-5-5-5"` (thus only having the final `'5'` left to process), we will have five parse stacks, as follows, where the top of the stack is shown to the right.



We see that the stacks are very similar, in fact some of them are equal. And since *shift* and *reduce* only depend on the top state of a stack, the processing function will do the same work for each of the five stacks.

To avoid this duplicated work, we change the type of parse stacks to a tree-structure, where the top states in the stacks are joined to a single "Tomita stack". This stack really represents a set of stacks – the set of all stacks with the same top element. The five different parse stacks of above will then be joined into one single Tomita stack.



The Tomita stack For more complicated grammars it might be the case that not all stacks have the same top state, which means that the processing function should work on a set of Tomita stacks. Since we have now changed data structure from a list of stacks to a set of Tomita stacks, we need new versions of the list functions we have been using. We can still use the list comprehensions on the sets, since the sets are ordered lists of Tomita stacks, but we need new versions of the concatenation functions ($+\!\!+$) and *concat*, which we will call *plusT* and *unionT*. We also need a way to join a list of Tomita stacks, which will be

[3]This is when we want all possible parses – the depth-first version is by far the fastest one if we only want the first parse result.

called *joinT*.

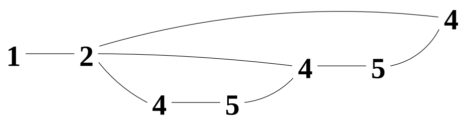
$$\begin{aligned} &\mathit{plusT} && :: \ \mathsf{Ord}\ c \Rightarrow \mathsf{TStacks}\ c \to \mathsf{TStacks}\ c \to \mathsf{TStacks}\ c \\ &\mathit{unionT} && :: \ \mathsf{Ord}\ c \Rightarrow [\mathsf{TStacks}\ c] \to \mathsf{TStacks}\ c \\ &\mathit{joinT} && :: \ \mathsf{Ord}\ c \Rightarrow [\mathsf{TStack}\ c] \to \mathsf{TStacks}\ c \end{aligned}$$

The type TStacks of sets of Tomita stacks will be implemented as ordered lists of Tomita stacks. But the exact definitions of the operations and the type TStack of Tomita stacks will be revealed in the next subsection; for now we can consider the Tomita stack as an abstract data type.

We also need new versions of the *push*, *top* and *pop* stack operations, which will be called *pushT*, *topT* and *popT*. The *popT* function will now return a list of stacks, since a single Tomita stack is really a representation of a collection of stacks.

$$\begin{aligned} &\mathit{pushT} && :: \ \mathsf{LRState}\ c \to \mathsf{TStack}\ c \to \mathsf{TStack}\ c \\ &\mathit{topT} && :: \ \mathsf{TStack}\ c \to \mathsf{LRState}\ c \\ &\mathit{popT} && :: \ \mathsf{Int} \to \mathsf{TStack}\ c \to [\mathsf{TStack}\ c] \end{aligned}$$

The reason why we call the stack a “Tomita stack” is that since Haskell shares common subparts of our tree structure, the tails of the stacks will be shared and we end up with the tree stored as a graph in memory. This graph structure corresponds to the graph structured stack used in the original Tomita algorithm.



Reducing and shifting Tomita stacks The auxiliary functions reduce_S and shift_S have to be slightly modified, so we rename them to $\mathit{reduceT}_S$ and shiftT_S. In principle we only have to replace concatenation with union and (*stack*:) with *plusT* [*stack*]. Apart from that we have to remember that *popT* returns a collection of stacks, not a single one.

$$\begin{aligned} &\mathit{reduceT}_S && :: \ \mathsf{TStack}\ c \to \mathsf{TStacks}\ c \\ &\mathit{reduceT}_S\ \mathit{stack} && = \ [\mathit{stack}]\ \text{‘}\mathit{plusT}\text{‘} \\ &&& \quad\ \mathit{unionT}\ [\mathit{reduceT}_S\ (\mathit{pushT}\ \mathit{state}\ \mathit{stack}')\ \mid \\ &&& \qquad\qquad (\mathit{cat},\ n) \leftarrow \mathit{reduce}\ (\mathit{topT}\ \mathit{stack}), \\ &&& \qquad\qquad \mathit{stack}' \leftarrow \mathit{popT}\ n\ \mathit{stack}, \\ &&& \qquad\qquad \mathit{state} \leftarrow \mathit{shift}\ (\mathit{topT}\ \mathit{stack}')\ \mathit{cat}] \end{aligned}$$

The *shiftT* function has to join the resulting list of Tomita stacks into a set.

$$\begin{aligned} &\mathit{shiftT}_S && :: \ \mathsf{TStack}\ c \to t \to \mathsf{TStacks}\ c \\ &\mathit{shiftT}_S\ \mathit{stack}\ \mathit{sym} && = \ \mathit{joinT}\ [\mathit{pushT}\ \mathit{state}\ \mathit{stack}\ \mid \\ &&& \qquad\qquad \mathit{cat} \leftarrow \mathit{terminal}\ \mathit{sym}, \\ &&& \qquad\qquad \mathit{state} \leftarrow \mathit{shift}\ (\mathit{topT}\ \mathit{stack})\ \mathit{cat}] \end{aligned}$$

Tomita processing The processing functions are not changed much from the breadth-first version. The only real difference is that concatenation is replaced by union.

```
processT                 ::  TStacks c → [ t ] → TStacks c
processT [ ]      input   =  [ ]
processT stacks input     =  shiftT stacks′ input
  where stacks′           =  unionT [ reduceT_S stack | stack ← stacks ]

shiftT                   ::  TStacks c → [ t ] → TStacks c
shiftT stacks [ ]         =  [ stack | stack ← stacks, accept (topT stack) ]
shiftT stacks (sym : input)
                          =  processT stacks′ input
  where stacks′           =  unionT [ shiftT_S stack sym | stack ← stacks ]
```

6.2.5 The Tomita parse stack

The Tomita parse stack will be a tree of states – the top state can be followed by any number of stacks, represented by a set of stacks.

```
data TStack c    =  LRState c ::: TStacks c
type TStacks c   =  [ TStack c ]
```

Remember that the set of stacks is a list of stacks, ordered by the top state. The reason why we do not use our standard Set datatype is that we cannot make use of set union (<++>), but have to define our own variant *plusT*, which also joins the tails of the stacks.

Stack operations To pop a number of states from a Tomita stack we have to collect all possible stacks that can be the result.

```
popT                       ::  Int → TStack c → [ TStack c ]
popT 0 stack                =  [ stack ]
popT n (state ::: stacks)   =  concat [ popT (n − 1) stack | stack ← stacks ]
```

To push a state onto a stack, we put it in front of the singleton stack. The top element of a Tomita stack is also straightforward.

```
pushT                    ::  LRState c → TStack c → TStack c
pushT state stack         =  state ::: [ stack ]

topT                     ::  TStack c → LRState c
topT (state ::: stacks)   =  state
```

Set operations To join a list of stacks into a set, we turn each of the stacks into a singleton set, which we then union together. This corresponds to the *makeSet* function on ordered lists in appendix A.

```
joinT      ::  Ord c ⇒ [TStack c] → TStacks c
joinT xs   =   unionT [[x] | x ← xs]
```

To union a list of sets together, we do the usual splitting and merging as in the ordered list union in the appendix. The merging is done by the *plusT* operation.

```
unionT                      ::  Ord c ⇒ [TStacks c] → TStacks c
unionT [ ]                  =   [ ]
unionT [x]                  =   x
unionT xys                  =   let (xs, ys) = split xys
                                in unionT xs `plusT` unionT ys
   where split [ ]          =   ([ ], [ ])
         split [x]          =   ([x], [ ])
         split (x : y : xys) =  let (xs, ys) = split xys in (x : xs, y : ys)
```

To merge two sets together, we go through them in parallel, similarly to the *mergeWith* function for association lists in the appendix. Since the tail of each Tomita stack is a set of Tomita stacks, we use *plusT* recursively to join the tails.

```
plusT          ::  Ord c ⇒ TStacks c → TStacks c → TStacks c
plusT [ ] ys   =   ys
plusT xs [ ]   =   xs
plusT xs@(x@(a ::: as) : xs′) ys@(y@(b ::: bs) : ys′)
               =   case compare a b of
                     LT → x : plusT xs′ ys
                     GT → y : plusT xs ys′
                     EQ → (a ::: plusT as bs) : plusT xs′ ys′
```

Exercise

The alert reader may have noticed that the type of sets of Tomita stacks corresponds to a finite map from states to sets of Tomita stacks.

```
newtype TStacks c  =  TS (Map (LRState c) (TStacks c))
```

We have to declare it as a **newtype**, since the type is recursive. How will the stack operations look like is we use this definition of Tomita stacks? What operations on finite maps can we reuse? How will the type TStack look like, and what changes must be made to the processing functions?

□

6.3 Adding parse results

For the parsing to be really useful we have to return some kind of parse result – the final parse stack doesn't give us much. This means that we have to change the type of stacks, as well as the processing functions.

6.3.1 The parse stack with interleaved parse results

It is only the states below the top state in a stack that have a parse result associated with them; i.e. the top state does not have a parse result yet. This suggests that we use a list of pairs of parse results and states for the *tail* of the stack, and then pair that list with the top of the stack.

$$\textbf{type}\ \mathsf{Stack}\ c\ res \quad = \quad (\mathsf{LRState}\ c,\ [\,(res,\ \mathsf{LRState}\ c)\,])$$

Stack operations The *push* function has to take a parse result as well as a state to be able to push something onto the stack. Observe that the given parse result is associated with the previous top state, not for the new state.

$$\begin{aligned}&push \quad :: \quad \mathsf{LRState}\ c \to res \to \mathsf{Stack}\ c\ res \to \mathsf{Stack}\ c\ res\\&push\ state\ result\ (state',\ stack) \quad = \quad (state,\ (result,\ state') : stack)\end{aligned}$$

The *top* of the stack is the first element of the pair.

$$\begin{aligned}&top &&:: \quad \mathsf{Stack}\ c\ res \to \mathsf{LRState}\ c\\&top\ (state,\ \_) &&= \quad state\end{aligned}$$

The new *pop* function will, apart from the new stack, also return the parse results that have been popped from the stack. This list might then be used while reducing to calculate a new parse result. The list will be returned in reversed order, since a stack stores its elements in reversed order. This reversing is done by accumulating the popped trees via an auxiliary *pop′* function.

$$\begin{aligned}&pop \quad :: \quad \mathsf{Int} \to \mathsf{Stack}\ c\ res \to ([\,res\,],\ \mathsf{Stack}\ c\ res)\\&pop\ n \ = \quad pop'\ n\ [\,]\\&\quad \textbf{where}\ pop'\ 0\ popped\ stack \quad = \quad (popped,\ stack)\\&\qquad\qquad pop'\ n\ popped\ (state,\ (result,\ state') : stack)\\&\qquad\qquad\qquad\qquad\qquad = \quad pop'\ (n-1)\ (result : popped)\ (state',\ stack)\end{aligned}$$

In the rest of this section we will consider parse trees as the possible parse results. An exercise in section 6.3.3 gives another alternative type of parse result.

6.3.2 Depth- and breadth-first parsing with parse trees

Reducing and shifting stacks The changes necessary to the $reduce_S$ and $shift_S$ functions are to calculate a parse tree which will be pushed onto the stack.

The parse tree for *reduce*$_S$ is the reduced category together with the popped parse trees.

```
reduce_S         ::  Stack c (ParseTree c t) → [Stack c (ParseTree c t)]
reduce_S stack   =   stack : concat [reduce_S (push state tree stack') |
                                     (cat, n) ← reduce (top stack),
                                     let (popped, stack') = pop n stack,
                                     let tree = cat :^ popped,
                                     state ← shift (top stack') cat]
```

The parse tree for *shift*$_S$ is the input symbol together with its corresponding category.

```
shift_S   ::  Stack c (ParseTree c t) → t → [Stack c (ParseTree c t)]
shift_S stack sym  =  [push state tree stack |
                        cat ← terminal sym,
                        let tree = cat :^ [Leaf sym],
                        state ← shift (top stack) cat]
```

Processing the input We do not have to change the processing functions for depth-first and breadth-first – the necessary changes are already done in *reduce*$_S$ and *shift*$_S$. Only the type signatures will be slightly different, to incorporate that the stacks now hold parse trees as results.

```
processDF  ::  Stack c (ParseTree c t) → [t] → [Stack c (ParseTree c t)]
shiftDF    ::  Stack c (ParseTree c t) → [t] → [Stack c (ParseTree c t)]

processBF  ::  [Stack c (ParseTree c t)] → [t] → [Stack c (ParseTree c t)]
shiftBF    ::  [Stack c (ParseTree c t)] → [t] → [Stack c (ParseTree c t)]
```

6.3.3 Tomita parsing

The changes to the Tomita parsing functions are the same as the changes to the breadth-first version above. The *processT* and *shiftT* functions remain the same; and the changes to *shiftT*$_S$ and *reduceT*$_S$ is that we have to calculate a parse tree to be pushed onto the stack.

Exercise

Change *reduceT*$_S$ and *shiftT*$_S$ in the same way as was done for *reduce*$_S$ and *shift*$_S$; by calculating a parse tree to be pushed onto the Tomita stack.

Finally implement the functions *processT* and *shiftT* to work on Tomita stacks with parse trees. You can assume the changes to the Tomita stack described below.

□

Exercise

Implement an alternative way to do the LR processing by building a chart while reducing the stacks. Recall from chapter 5 that a chart is a collection of edges, and in this case it will only consist of passive edges, where a passive edge is a triple $\langle i, j : A\rangle$, saying that the category A spans the nodes i and j.

$$\textbf{type}\ \mathsf{Passive}\ c \quad = \quad (\mathsf{Int},\ \mathsf{Int},\ c)$$

Our stack will now have passive edges as results, and not parse trees. When reducing a stack, we add a new passive edge to the chart. The category of the passive edge is the reduced category, and the starting node can be recovered from the edges that are popped off the stack. The ending node depends on where in the input we are right now, which means that we have to propagate an integer saying how many symbols that have been processed so far.

$$\mathit{reduceT}_S \quad :: \quad \mathsf{TStack}\ c\ (\mathsf{Passive}\ c) \to \mathsf{Int} \to \mathsf{TStacks}\ c\ (\mathsf{Edge}\ c)$$

When shifting, we add a passive edge corresponding to the shifted category, spanning i and $i + 1$, where i is node of the current input symbol.

$$\mathit{shiftT}_S \quad :: \quad \mathsf{TStack}\ c\ (\mathsf{Passive}\ c) \to t \to \mathsf{Int} \to \mathsf{TStacks}\ c\ (\mathsf{Edge}\ c)$$

The processing function can now add all newly created edges to the final chart, before continuing with the next input symbol

```
$processT$    ::  $\mathsf{TStacks}\ c\ (\mathsf{Passive}\ c) \to [\,t\,] \to \mathsf{Int} \to \mathsf{Set}\ (\mathsf{Passive}\ c)$
$processT\ stacks\ input\ i$   =  $edges <\!\!+\!\!+\!\!> shiftT\ stacks'\ input\ i$
  where $stacks'$          =  $unionT\ [\,reduceT_S\ stack\ i \mid stack \leftarrow stacks\,]$
        $edges$            =  $makeSet\ [\,edge \mid stack' \leftarrow stacks',$
                                       $([\,edge\,],\ \_) \leftarrow popT\ 1\ stack'\,]$
```

The *shiftT* function can return the empty set when there is no input left, since we have already added all possible edges to the chart.

```
$shiftT$    ::  $\mathsf{TStacks}'\ c \to [\,t\,] \to \mathsf{Int} \to \mathsf{Set}\ (\mathsf{Passive}\ c)$
$shiftT\ stacks\ [\,]$             $\_$  =  $emptySet$
$shiftT\ stacks\ (sym : input)\ i$   =  $processT\ stacks'\ input\ (i+1)$
  where $stacks'$                 =  $unionT\ [\,shiftT_S\ stack\ sym\ i \mid$
                                        $stack \leftarrow stacks\,]$
```

When the processing has suceeded we can use the function *buildTrees* in section 5.4 to calculate the final parse trees. □

6.3.4 The Tomita stack

The change necessary for the Tomita stack is to pair a parse result with a set of stacks, for the *tail* of the Tomita stack, similar to what was done for the standard parse stacks. This means that the tail of a Tomita stack will be a finite map from results to sets of Tomita stacks.

```
data TStack c res   =  LRState c ::: Map res (TStacks c res)
type TStacks c res  =  [TStack c res]
```

Stack operations The change to *pushT* is very small; the pushed parse tree is paired with the old stack. The *topT* function remains the same.

```
pushT   ::  LRState c → res → TStack c res → TStack c res
pushT state result stack  =  state ::: (result |→ [stack])

topT    ::  TStack c res → LRState c
topT (state ::: stacks)  =  state
```

To pop elements off a Tomita stack, we use an accumulator, as we did for the ordinary parse stack. We only have to remember that there can be many resulting stacks, which gives a list of results.

```
popT      ::  Int → TStack c res → [([res], TStack c res)]
popT n    =   popT' n [ ]
  where popT' 0 popped stack  =  [(popped, stack)]
        popT' n popped (state ::: smap)
             = concat [popT' (n − 1) (result : popped) stack  |
                         (result, stacks) ← assocs smap,
                         stack ← stacks]
```

Set operations The *joinT* and *unionT* operations remain exactly as before, so we do not repeat their definitions here. The *plusT* will be just slightly different, in that we have to merge the two finite maps. This has the effect that the result type must have an ordering.

```
plusT  ::  (Ord c, Ord res) ⇒ TStacks c res → TStacks c res → TStacks c res
plusT [ ] ys   =   ys
plusT xs [ ]   =   xs
plusT xs@(x@(a ::: as) : xs') ys@(y@(b ::: bs) : ys')
              =   case compare a b of
                    LT → x : plusT xs' ys
                    GT → y : plusT xs ys'
                    EQ → (a ::: mergeWith plusT as bs) : plusT xs' ys'
```

Exercise

Implement the type of sets of Tomita stacks as a finite map from states to finite maps from results to Tomita stacks.

```
newtype TStacks c res  =  TS (Map (LRState c) (Map res (TStacks c res)))
```

What changes must be made to the stack operations and to the processing functions?

□

6.4 Initializing and running the parsing

The processing functions defined above all take as input a stack (or a collection of stacks) and return a collection of stacks. The main parsing function initializes the processing a single stack consisting of only the starting LR state. After the end of the processing we will have a collection of stacks, which all are of length 2 – the top state being the accepted state, and the bottom state being the starting state. Between those two states we will find one of the parse trees we are looking for. So to retrieve the final parse trees, we only have to pop one state off each of the stacks and use the popped parse tree.

Depth-first parsing The parsing function for the depth-first version uses the initial stack with the start state as the top element. All the previously mentioned functions (except for the stack operations) must be declared locally, because they all depend on the grammar.

```
parseDF  ::  Ord c ⇒ Grammar c t → [t] → [ParseTree c t]
parseDF grammar input  =  [tree |
                            stack ← processDF (startState, []) input,
                            let ([tree], _) = pop 1 stack]
  where processDF      =  ...
        shift          =  ...
        reduce         =  ...
        accept         =  ...
        startState     =  ...
```

Breadth-first parsing The single difference with the breadth-first version is that we start with a singleton list of initial stacks.

```
parseBF  ::  Ord c ⇒ Grammar c t → [t] → [ParseTree c t]
parseBF grammar input  =  [tree |
                            stack ← processBF [(startState, [])] input,
                            let ([tree], _) = pop 1 stack]
  where processBF      =  ...
        ...
```

Tomita parsing The Tomita version is similar, the only real issue being to remember that the *popT* operation returns a list of results instead of just a single result. Also we need an ordering on the terminals, since we use parse trees as parse results in the stack.

```
parseT               :: (Ord c, Ord t) ⇒ Grammar c t → [t] → [ParseTree c t]
parseT grammar input  =  [tree |
                             stack ← processT [startState ::: [ ]] input,
                             ([tree], _) ← popT 1 stack]
   where processT     =  ...
         ...
```

6.5 Creating the LR parsing table

There is a restriction on the grammar for the parsing process to be correct. There has to be only one production for the starting category, and that production has to be a unit production. The example grammar obeys this restriction by the unit production Sent $\longrightarrow$ Expr.

A grammar that doesn't obey the restriction can easily be augmented with a new starting category S' and a new production $S' \to S$, where S is the old starting category. In the literature one usually augments the grammar with a production $S' \to S\$$, where \$ is a new end-marker terminal category; but this is strictly only necessary when we have lookahead. See also the final exercise in this chapter.

The reason for the restriction is that otherwise will the final parsing stack not be of length 2 with the final parse tree in between as described previously.

The LR state The LR state is a set of LR items, an item is a dotted production. A dotted production is of the form $A \to \alpha \cdot \beta$, where the symbols after the dot represents the unrecognized part of the production. Since the symbols before the dot are uninteresting for the LR item, we skip that part. An item is therefore a pair of the sequence of categories β and what production the item is derived from.

```
type LRState c  =  Set (Item c)
type Item    c  =  ([c], Production c)
```

We will need a helper function for building a LR state from an initial set of items. This uses the *limit* function to calculate a minimal fixpoint set, and for each added item it adds new items for the next category of that item.

```
buildState                ::  Set (Item c) → LRState c
buildState                 =  limit more
   where more (cat : _, _) =  newItems cat
         more _            =  emptySet
```

The *newItems* function selects the productions for a given category.

```
newItems       ::  c → Set (Item c)
newItems cat   =   makeSet [(tofind, rule) |
                            rule@(cat′, tofind) ← elems productions,
                            cat == cat′]
```

The shift table The transition function finds all items in the given state matching the given category, and builds a new state from them.

```
nextState             ::  LRState c → c → LRState c
nextState state cat   =   buildState
                          (makeSet [(tofind, rule) |
                                    (cat′ : tofind, rule) ← elems state,
                                    cat == cat′])
```

If a transition is not possible, the resulting state will be an empty set, so the *shift* function has to check that the result is not an empty set.

```
shift              ::  LRState c → c → [LRState c]
shift state cat    =   if isEmpty state′ then [ ] else [state′]
  where state′     =   nextState state cat
```

The reduce table It is possible to reduce in a state if there is an item which has no more categories to find. The result will be the main category of the production and the number of symbols in the right-hand side, which determines the number of states to pop off the stack.

```
reduce         ::  LRState c → [(c, Int)]
reduce state   =   [(cat, length rhs) | ([ ], (cat, rhs)) ← elems state]
```

The accepting states A state is accepted if the starting category is recognized.

```
accept         ::  LRState c → Bool
accept state   =   or [cat == start | ([ ], (cat, _)) ← elems state]
```

The starting state For the starting state we only have to build the LR state corresponding to the starting category.

```
startState  ::  LRState c
startState  =   buildState (newItems start)
```

In figure 6.2 we show the LR automaton for the example grammar, with the states represented as sets of LR items.

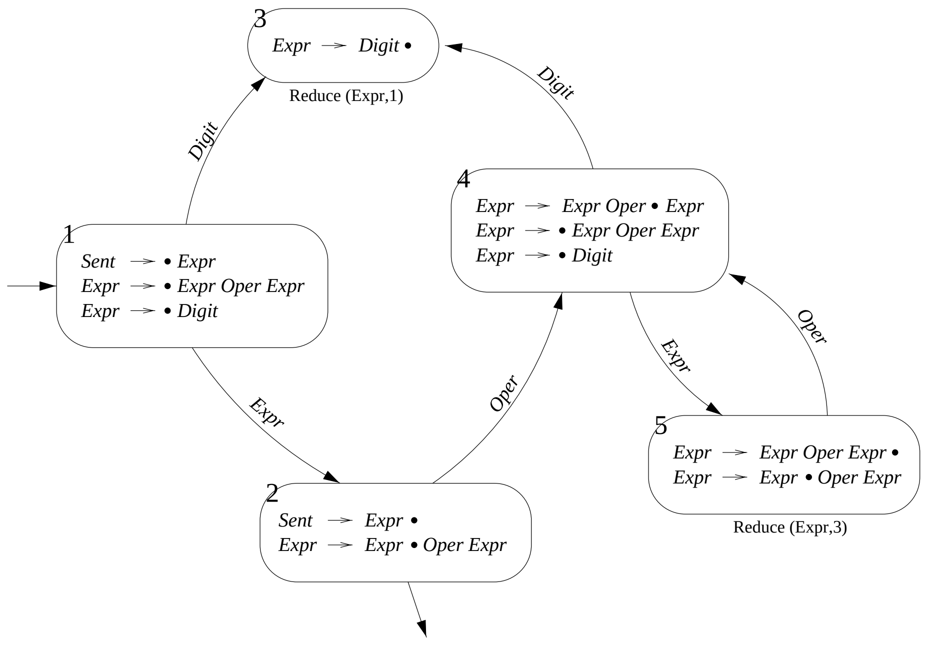


Figure 6.2: The LR automaton for the expression grammar.

Exercise

Make the LR functions more efficient by numbering all the sets of items, and using the index of the set as the LR state. Then you can create tables in the form of finite maps or arrays, to efficiently lookup the results of *shift*, *reduce* and *accept*.

Exercise

The implementation in this section uses an LR(0) automaton, with no lookahead for the reduction step. Implement a SLR(1) parser by the following changes.

- The *reduce* function has to take the next input category as another argument. In the definition for *reduce*, we can add the constraint that the next category is in the follow set of the reduced category (*nextcat*‘*elemSet*‘ *follow cat*).
- The grammar has to be augmented with a special end marker category E (and corresponding terminal), and the production for the starting category has to be changed to $S' \to S\ E$.
- In the parsing function, the end marker terminal has to be added to the end of the input sequence (*input* ++ [*endMarker*]).
- The $reduce_S$ function and corresponding has functions to be changed to accomodate the changed type of *reduce*.

□

6.6 Discussion

Space complexity The depth-first and breadth-first parsing strategies are both exponential in the worst case. The number of stacks can grow exponentially for an ambigous grammar such as the expression grammar.

By using the Tomita stack, we will not get this explosion of stacks, since the top states are merged together. The data structure TStack is a tree; but since whenever we duplicate a stack the results will be shared in a lazy language, this means that the bottoms of the tree of stacks will be shared. Thus, the Tomita stack will be stored in memory as a directed acyclic graph. This memory structure makes the memory behaviour the same as Tomita's original GLR algorithm [42]. But in the version presented here we do not have to implement a data structure of directed graphs, but a simpler tree data structure.

Time complexity Unfortunately the time complexity is not the same as in the GLR algorithm. Although the stacks are are stored in memory polynomial in the length of the input, Haskell doesn't know this. It is still possible to do the same reduction on shared sub-stacks twice (or more times), because we cannot observe that they are the same stack.

The time complexity of the original GLR algorithm is $O(n^{\delta+1})$, where δ is the length of the longest production in the grammar. Our approximation will still be in the worst case exponential in the length of the input, which is the same as the depth-first and breadth-first versions. The Tomita approximation is still an improvement over the other versions, although not complexity-wise.

Implicit stack graph Another problem with the implicit sharing of sub-stacks is that it is impossible to extract the graph if we e.g. would like to visualize it in some way. Or to be more correct, it is impossible unless we use some impure extension of Haskell, such as observable sharing described in chapter 8.

Limitations on the grammar Tomita's original algorithm does not work on hidden left-recursive grammars. This is because we get infinitely many reductions when trying to decide how many times we want to apply the left-recursion. The problem has been solved in e.g. [30, 31] by tweaking the data structures in different ways. The implementations in this chapter still has a problem with hidden left-recursive grammars though.

Lazy evaluation Laziness is not used at all in the parsing functions – they are very strict indeed. If we are interested in only the first few parse trees, we will gain something by lazy evaluation. Especially the depth-first version will gain a lot on using laziness, since it will only create one parse result at the time.

Another place where we could make use of laziness would be in the exercise in section 6.5, which was about converting the LR states to integers, making the LR tables more efficient.

Chapter 7

CYK Parsing

The parsing algorithm of Cocke, Younger and Kasami [19, 47] is one of the oldest parsing algorithms known. Still it is a very nice algorithm due to its simplicity. In this chapter we will use vectors to represent the parse matrix, and compute new cells by repeated scalar products. The drawback with the CYK parsing algorithm is that it only works for context-free grammars in Chomsky normal form.

The last parsing algorithm in this thesis is CYK parsing, which can be implemented very compact and elegant in a functional way, while still being as efficient as the standard imperative algorithms. It is one of the earliest context-free parsing algorithms, named after its inventors Cocke, Younger and Kasami [19, 47].

This algorithm only works for context-free grammars in Chomsky normal form, which is no serious problem since every context-free grammar can be transformed to Chomsky normal form, as discussed in section 2.1 and in the exercise in section 2.3. A grammar is in Chomsky normal form if all rules are either unit productions to terminals ($C \to s$), or productions with two non-terminals on the right-hand side ($C \to AB$).

7.1 CYK parsing

In CYK parsing one uses a parse matrix instead of a chart. One can see it as instead of having Earley states consisting of edges that end in a certain node, we divide the chart further into cells of edges that both start and end in given nodes. This turns the chart into a matrix of cells, where we will write $\mathbf{C}_{i,j}$ for the i:th cell in the j:th Earley state. Furthermore we will only store passive edges in the cells, which means that a cell only has to consist of a set of categories.

The parse matrix will thus consist of the cells $\mathbf{C}_{i,j}$, with $0 \leqslant i < j \leqslant n$ where n is the length of the input sequence. Observe that the inequality $i < j$ is strict, since we have no empty productions. Each cell $\mathbf{C}_{i,j}$ is a set that contains all the categories spanning the nodes i to j. Or using the vocabulary of chart parsing, $\mathbf{C}_{i,j} = \{ C \mid \langle i, j : C \rangle \}$.

7.1.1 Initialization of the parse matrix

Since the grammar is in Chomsky normal form, there are no empty productions. This means that an application of a binary production $C \to AB$ will span at least two nodes (since A and B span at least one node each). So, the only way to fill the cells $\mathbf{C}_{i-1,i}$ is to use the terminal productions $C \to t_i$, where t_i is the i:th input terminal. Before we start the real parsing process we initialize these cells by applying the appropriate terminal productions.

$$\mathbf{C}_{i-1,i} \quad = \quad \{ C \mid C \to t_i \}$$

7.1.2 The CYK parsing process

In a grammar in Chomsky normal form, a category C between i and k, where $k - i \geqslant 2$, must come from a binary production $C \to AB$. Then the category A must span the nodes i and j for some $i < j < k$,[1] and the category B must span the nodes j and k. Therefore we can calculate the value of $\mathbf{C}_{ik}$ as the union of all $\mathbf{C}_{ij}\mathbf{C}_{jk}$ for $i < j < k$, where the cell product is defined as follows.

$$\mathbf{AB} \quad = \quad \{ C \mid A \in \mathbf{A},\, B \in \mathbf{B},\, C \to AB \}$$

And if we use set union as the additive operator, this can be seen as the scalar product of a row vector and a column vector.

$$\begin{aligned} \mathbf{C}_{ik} &= \bigcup_{j=i+1}^{k-1} \mathbf{C}_{ij}\mathbf{C}_{jk} \\ &= \langle \mathbf{C}_{i,i+1},\, \mathbf{C}_{i,i+2} \ldots \mathbf{C}_{i,k-1} \rangle \cdot \langle \mathbf{C}_{i+1,k},\, \mathbf{C}_{i+2,k} \ldots \mathbf{C}_{k-1,k} \rangle \end{aligned}$$

One way to see this is as calculating the transitive closure of a $(n+1) \times (n+1)$ matrix, which is empty everwhere except for the cells just above the main diagonal. Valiant uses this in [43] to translate the CYK parsing problem into matrix multiplication to show that it is possible to recognize general context-free languages in less than cubic time.

7.1.3 An illustrative example

As an example grammar we use the same grammar as we used in chapter 5, but converted to Chomsky normal form. This means that the two unit productions

[1] The inequality $i < j < k$ is strict since there are no empty productions.

are deleted, but instead the Noun and Verb terminals have to be repeated in the NP and VP definitions.

$$
\begin{array}{lllllllll}
\mathsf{S} & \longrightarrow & \mathsf{NP}\ \ \mathsf{VP} & & & & & & \\
\mathsf{VP} & \longrightarrow & \mathsf{Verb}\ \ \mathsf{NP} & & & & & & \\
 & | & \textit{flies} & | & \textit{like} & | & \ldots & & \\
\mathsf{NP} & \longrightarrow & \mathsf{Det}\ \ \mathsf{Noun} & & | & & \mathsf{NP}\ \ \mathsf{PP} & & \\
 & | & \textit{flies} & | & \textit{time} & | & \textit{arrow} & | & \ldots \\
\mathsf{PP} & \longrightarrow & \mathsf{Prep}\ \ \mathsf{NP} & & & & & & \\
\mathsf{Verb} & \longrightarrow & \textit{flies} & | & \textit{like} & | & \ldots & & \\
\mathsf{Noun} & \longrightarrow & \textit{flies} & | & \textit{time} & | & \textit{arrow} & | & \ldots \\
\mathsf{Det} & \longrightarrow & \textit{an} & | & \ldots & & & & \\
\mathsf{Prep} & \longrightarrow & \textit{like} & | & \ldots & & & &
\end{array}
$$

In figure 7.1 the parse matrix for the sentence “time flies like an arrow” is shown, just before calculating the contents of cell $\mathbf{C}_{15}$.

$$
\begin{aligned}
\mathbf{C}_{15} &= \mathbf{C}_{12}\cdot\mathbf{C}_{25} \cup \mathbf{C}_{13}\cdot\mathbf{C}_{35} \cup \mathbf{C}_{14}\cdot\mathbf{C}_{45} \\
&= \{\mathsf{Noun,Verb,NP,VP}\}\cdot\{\mathsf{PP}\} \cup \{\mathsf{S}\}\cdot\{\mathsf{NP}\} \cup \emptyset\cdot\{\mathsf{Noun,NP}\} \\
&= \{\mathsf{S,VP}\}
\end{aligned}
$$

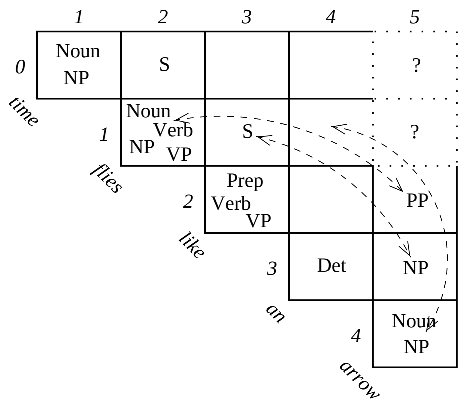


Figure 7.1: The parse matrix before calculation of cell $\mathbf{C}_{15}$.

7.2 A Haskell implementation of CYK parsing

For the sake of presentation we define all functions in this section on the top-level, but in reality they are all local definitions to the main parsing function. This means that we can assume the grammar to be known, and we will use the names *terminal* and *productions* for the terminal function and the set of production rules.

$$\begin{array}{lll} \textit{terminal} & :: & t \rightarrow \mathsf{Set}\ c \\ \textit{productions} & :: & \mathsf{Set}\ (\mathsf{Production}\ c) \end{array}$$

Observe that since the grammar is in Chomsky normal form, every production is a binary production $(c,\ [\,a,\ b\,])$.

7.2.1 The parse matrix

As described above, a cell constists of a set of categories.

$$\textbf{type}\ \mathsf{Cell}\ c \quad = \quad \mathsf{Set}\ c$$

To calculate the elements in the cells we need to have a representation of a vector of cells, which we can do the scalar product over. Since for practical grammars (both for formal and natural languages), most of the cells will be empty, we use a sparse representation of the vectors. A vector is thus an ordered list of pairs of indices and cells.

$$\textbf{type}\ \mathsf{Vector}\ c \quad = \quad [\,(\mathsf{Int},\ \mathsf{Cell}\ c)\,]$$

Indices start from 0, which means that the list $[\,(1,\ a),\ (3,\ b)\,]$ will be a representation of the vectors $\langle \emptyset, a, \emptyset, b \rangle$, $\langle \emptyset, a, \emptyset, b, \emptyset \rangle$, $\langle \emptyset, a, \emptyset, b, \emptyset, \emptyset \rangle$, etc.

The representation of the parse matrix will be a list of row vectors $\mathbf{V}_i$, representing all the edges starting in the node i. The maximum index k is the same as the number of input tokens.

$$\mathbf{V}_i \quad = \quad \langle \mathbf{C}_{i,i+1}, \mathbf{C}_{i,i+2}, \ldots, \mathbf{C}_{i,k} \rangle$$

To be able to calculate the scalar product over $\mathbf{V}_i$, we will also need the column vector $\langle \mathbf{C}_{i+1,k} \ldots \mathbf{C}_{k-1,k} \rangle$, but this will be created while updating the row vectors, and will not have to be remembered afterwards.

7.2.2 Scalar product over vectors

To compute the scalar product of two vectors we go through them at the same time, comparing their respective indices. If they are equal, we multiply the cells, adding to the result of the rest of the computation. Otherwise, we just move further on.

```
scalarProduct          :: Ord c ⇒ Vector c → Vector c → Cell c
scalarProduct [ ] _    =  [ ]
scalarProduct _ [ ]    =  [ ]
scalarProduct as@((i, a) : as') bs@((j, b) : bs')
                       =  case compare i j of
                            LT → scalarProduct as' bs
                            GT → scalarProduct as bs'
                            EQ → scalarProduct as' bs' <++> product a b
```

To multiply two cells, we try to find a binary production whose right-hand side categories are contained in the given cells.

```
product           ::  Cell c → Cell c → Cell c
product as bs     =   makeSet [ c | (c, [ a, b ]) ← elems productions,
                                    a `elemSet` as,
                                    b `elemSet` bs ]
```

Exercise

Implement the *product* using a finite map for the productions in the grammar. □

7.2.3 Processing one input token

When we have processed k input tokens, we will have a list of k intermediate row vectors $\mathbf{V}_i = \langle \mathbf{C}_{i,i+1} \ldots \mathbf{C}_{i,k} \rangle$. The next input token will create a new row vector $\mathbf{V}_{k+1} = \langle \mathbf{C}_{k,k+1} \rangle$ consisting of the categories for the given terminal. The previous vectors $\mathbf{V}_i$ also have to be updated with a new cell $\mathbf{C}_{i,k+1}$ at the end.

```
nextInputToken      ::   (Int, [Vector c]) → t → (Int, [Vector c])
nextInputToken (size, vectors) token   =   (size', vectors')
  where size'       =    size + 1
        vectors'    =    [ (size', cell) ]
                    :    updateVectors vectors [ (size, cell) ] size size'
        cell        =    terminal token
```

The *size* variable is a counter of how many cells have been processed so far, i.e. the number of vectors.

To calculate the new row vector $\langle \mathbf{C}_{i,i+1} \ldots \mathbf{C}_{i,k}, \mathbf{C}_{i,k+1} \rangle$ from the old one $\langle \mathbf{C}_{i,i+1} \ldots \mathbf{C}_{i,k} \rangle$, we have to add the cell $\mathbf{C}_{i,k+1}$. That cell is calculated as the scalar product of the old row vector and the column vector $\langle \mathbf{C}_{i+1,k} \ldots \mathbf{C}_{k-1,k} \rangle$. So the second argument to *updateVectors* is the column vector, which is updated for each new row vector. We also have to remember the row and column indices.

Remember that it is only when the scalar product is non-empty that we update the row and column vector.

```
updateVectors   ::  [Vector c] → Vector c → Int → Int → [Vector c]
updateVectors [ ] _ _ _  =  [ ]
updateVectors (row : rows) col nrow ncol
      | isEmpty product   =  row  : updateVectors rows col  nrow′ ncol
      | otherwise         =  row′ : updateVectors rows col′ nrow′ ncol
  where product           =  scalarProduct row col
        nrow′             =  nrow − 1
        row′              =  row ++ [(ncol, product)]
        col′              =  (nrow′, product) : col
```

7.2.4 Processing the whole input

The main processing function takes as input a list of terminals, and gives as output one single cell, which is the cell spanning the whole sentence.

We have to iterate the *nextInputToken* on all the input, building the final list of vectors. The last vector in this list is the vector $\langle \mathbf{C}_{0,1}, \mathbf{C}_{0,2} \ldots \mathbf{C}_{0,n} \rangle$, where n is the length of the sentence (which is equal to the *size* returned with the vectors). It is the last cell $\mathbf{C}_{0,n}$ that we want to return, but since the vectors are stored in a sparse way we have to check that the last cell in the last vector is indeed $\mathbf{C}_{0,n}$, which is done by comparing the length of the sentence with the index of the last cell.

```
process   ::   [t] → Cell c
process input
      | size == ncell      =  cell
      | otherwise          =  emptySet
  where (size, vectors)    =  foldl nextInputToken (0, [ ]) input
        (ncell, cell)      =  last (last vectors)
```

To recognize an input sequence we only have to check whether the starting category is contained in the final cell.

```
recognize   ::   Ord c ⇒ Grammar c t → [t] → Bool
recognize (_, start, terminal, productions) input
                           =  start ‘elemSet‘ process input
  where process            =  ...
        nextInputToken     =  ...
        updateVectors      =  ...
        scalarProduct      =  ...
        product            =  ...
```

Exercise

Implement a CYK recognizer for general context-free grammars, by translating the grammar to Chomsky normal form. The translation is described in an exercise in section 2.3.

□

7.3 Calculating the parse trees

To make things more interesting, we also add calculation of parse trees to the algorithm. To do this we change the representation of a cell to a finite map from a category to its parse trees.

```
type Cell c t  =  Map c [ParseTree c t]
```

Things will be simplified if we define an auxiliary lookup function which always succeeds, indicating failure by the empty list.

```
(??)       ::  Ord c ⇒ Cell c t → c → [ParseTree c t]
cell ?? c   =  case cell ? c of
                 Just trees → trees
                 Nothing    → [ ]
```

The functions *nextInputToken*, *scalarProduct*, *updateVectors* and *process* remain almost the same. We will only have to use the functions *emptyMap* instead of *emptySet* (in *process*), *isEmptyMap* instead of *isEmpty* (in *updateVectors*), *joinCell* instead of (<++>) (in *scalarProduct*), and *terminalCell* instead of *terminal* (in *nextInputToken*). The last two functions are new functions described in this section, together with the new cell *product*.

To join two cells we just merge the maps using list concatenation.

```
joinCell  ::  Ord c ⇒ Cell c t → Cell c t – Cell c t
joinCell  =   mergeWith (++)
```

We need a new terminal lookup function *terminalCell* which has to return a finite map instead of a set. To do this we pair each category with its corresponding parse tree. Observe that the association list is ordered since the original *terminal* function returns a set, and therefore we can use the *ordMap* function instead of *makeMap*.

```
terminalCell              ::  t → Cell c t
terminalCell term         =   ordMap [(cat, treesFor cat) |
                                          cat ← elems (terminal term)]
  where treesFor cat      =   [cat :^ [Leaf term]]
```

The new cell product has to look up the parsetrees in the given cells and combine them to new trees.

```
product             ::  Cell c t → Cell c t → Cell c t
product acell bcell  =  makeMapWith (++)
                        [ (c, [ c :^ [ atree, btree ] ]) |
                          (c, [ a, b ]) ← elems productions,
                          atree ← acell ?? a,
                          btree ← bcell ?? b ]
```

Finally we can define a parsing function that takes a list of terminals, and returns a collection of parse trees. It only has to lookup the starting category in the final cell from the processing.

```
parse  ::  Ord c ⇒ Grammar c t → [ t ] → [ ParseTree c t ]
parse (_, start, terminal, productions) input
                                  =  process input ?? start
    where process                 =  ...
          nextInputToken          =  ...
          updateVectors           =  ...
          scalarProduct           =  ...
          terminalCell            =  ...
          product                 =  ...
```

Exercise

Extend the previuos exercise by implementing CYK parsing for general context-free grammars, not just recognition. To do this you have to transform the resulting parse trees after successful parsing, to correspond to the original grammar instead of the Chomsky normal form variant.

Exercise

Instead of calculating the parse trees while parsing, it is possible to do it afterwards from the parse matrix. Translate the matrix to a chart (a list of passive edges), so that you can apply the *buildTrees* function of section 5.4 to calculate the parse trees.

```
type Passive c = (Int, Int, c)
```

□

7.4 Discussion

Space complexity In the simple parse matrix, every cell can have at most $|N|$ elements, where N is the set of non-terminals in the grammar. And since the matrix is two-dimensional, the space complexity is $O(n^2|N|)$, where n is the number of input tokens.

The final parse matrix also holds parse trees. And since there can be an exponential number of trees, the space complexity will be exponential in the length of the input.

Time complexity The cell product is defined by a list comprehension over the productions in the grammar, giving $|G|$ possible choices of categories. Each of these categories has to first be checked for inclusion in the given cells, and be turned into a set, amounting to an extra logarithmic factor. Therefore, the cell product as defined in section 7.2.2 has a worst-case complexity of $O(|G| \log |G|)$. By implementing the product in a different way, it is possible to get rid of the logarithmic factor and thus have an $O(|G|)$ worst-case complexity.

The scalar product over two vectors of length k consists of k cell products together with k unions, in the worst case. To build an arbitrary cell, we have to do a scalar product over two vectors of length $k \leqslant n$. The $O(n)$ cell products can be calculated in $O(n|G| \log |G|)$ time, and the $O(n)$ union in $O(n|N|)$ time. Since $|N| \leqslant |G|$, the unions are outweighed by the products, and the complexity for building an arbitrary cell is $O(n|G| \log |G|)$.

This gives us a final worst-case complexity of $O(n^3|G| \log |G|)$ for CYK parsing without calculating parse trees. If we optimize the cell product, we can also remove the logarithmic factor. Observe that calculating the parse trees is still exponential in the length of the input.

Comparing with chart parsing The worst-case complexity of chart parsing is $O(n^3|G|^2)$, as discussed in section 5.7.[2] CYK parsing is linear in the size of the grammar, while chart parsing is quadratic. So why don't we simply convert our general grammar G into a Chomsky normal form equivalent G' and parse using CYK, to get an improvement of the complexity? This is because the size of G' might be quadratic in the size of G, or to put it another way $O(|G'|) = O(|G|^2)$. Thus we will still get quadratic complexity in the size of the original grammar G.

Lazy evaluation The implementation presented here does not make use of laziness much. If a cell is a "dead end" it will never be calculated – i.e. if no later cell will use the contents of the dead cell. This will happen only when the dead cell will only be combined with empty cells – then we know that the cell product will be empty and do not have to look into the dead cell.

The same kind of reasoning goes for the parse trees – only the parse trees that are really used in the final result will be calculated. Coming back to our example grammar, this means that the parse tree of the sentence (S) "flies like an arrow" will never be calculated, only the parse tree of the corresponding verb phrase (VP).

[2]We have removed the factor δ^2, since it is not interesting for this discussion.

Chapter 8

Grammar-collecting Parser Combinators

With the help of an extension of Haskell called observable sharing, introduced by Claessen and Sands [6], we define context-free parser combinators that can handle general context-free grammars, even left-recursive ones. We can also parse with any available context-free parsing algorithm, and not just recursive descent parsing, which all standard parser combinators implement. We do this by collecting the context-free productions that correspond to the structure of the parser, and invoking any parsing algorithm, such as chart parsing, LR parsing or CYK parsing.

There are in principle three different ways in which one can define a parser for a given grammar. All three have their own specific advantages and disadvantages.

Explicit grammar object One approach is to implement the grammar as an explicit object in the host language, and then use a nice parsing algorithm to parse input sentences. This approach is treated in chapters 5, 6 and 7.

The main advantage is that there are no restrictions on the parsing algorithm, which can be tailor-made for our purposes.

The main disadvantage is that we cannot parse directly to the semantic result type we want, but have to go through an intermediate type such as the type of parse trees or charts.

Parser combinators Another approach is to implement the grammar directly with parser combinators, and use a suitable parser type. This approach is treated in chapters 3 and 4. Parser combinators form a domain-specific embedded language [14], with all accompanying features.

The three main advantages are that *i*) we can use the features of the host language to create new combinators; *ii*) the parsers can return the desired semantic action directly instead of using an intermediate type of parse results; and *iii*) the type checker in the host language can catch errors in the grammar, which can be very difficult to find with an untyped grammar. Furthermore, many parser types can be used to implement monadic combinators, which makes it possible to parse non context-free languages.

The main disadvantage with parser combinators is that they all implement the same parsing algorithm – recursive-descent parsing, which is not the most efficient parsing algorithm for many kinds of grammars. The algorithm is not even capable of parsing all possible grammars – left-recursive grammars leads to non-termination, which is a seriuos problem since they are very common in both formal and natural languages.

Parser generator A third alternative is to use a parser generator, such as Happy [28]. A parser generator extends the host language with the ability to define a grammar, together with semantic actions. The grammar file is then compiled to a tailor-made parser in the host language.

This approach combines many of the advantages of the previous two approaches – we can use an efficient parsing algorithm, we can define semantic actions on grammar productions directly, and the semantic results are type checked by the host language.

The main disadvantage is that we cannot use features of the host language to define "macros" for simplifying the grammar writing. Another disadvantage is that we have to learn a new syntax for writing grammar files, but this disadvantage can be considered minor since the syntax often is very natural and simple.

Combining combinators with explicit grammars

In this chapter we show how to combine the first two approaches by using an impure extension of Haskell. Claessen and Sands [6] have investigated the idea of observable sharing for hardware circuit descriptions, and we will use this idea to implement parser combinators which can be used to implement any parsing algorithm, not just recursive descent.

Our basic idea is that if we have a way of detecting when a parser is a subelement of itself, i.e. that we have some kind of cycle or recursion in the grammar, then we can collect the call structure of the parser as a set of productions in a context-free grammar. After that we simply call the parsing algorithm of our choice with the grammar and the input. This means that we translate a grammar as a Haskell program (defined by parser combinators) into a grammar as a Haskell object (as a set of production rules), parse using this grammar object, and finally we translate back the result.

8.1 Observable sharing

Claessen and Sands introduce in [6] an extension of Haskell called *observable sharing*, which allows them to detect and manipulate cycles in data structures. They use this for hardware circuit descriptions, and we will use it to collect the underlying grammar of a combinator parser.

A problem with observable sharing is that it is not a conservative extension of Haskell. In their paper Claessen and Sands investigate what properties of Haskell are lost and what remains, and it turns out that the extension is indeed small:

> We have shown that the extended language has a rich equational theory, which means that the semantics is robust with respect to program transformations which respect sharing properties. For example, we have shown that standard compiler transformations which use strictness analysis to turn call-by-need into call-by-value are still sound in this extension. [6]

The extension introduces an abstract data type, the type of references to objects, and three new functions to create references, recover the values, and compare references for equality.

```
newtype Ref α  =  ...

instance Eq (Ref α)

ref    ::  α → Ref α
deref  ::  Ref α → α
```

8.1.1 Example implementation of observable sharing

One way to implement the extension in Haskell is to use *unsafePerformIO* and IO references as described by Peyton-Jones [34], but then one must be careful to turn off any inlining and common subexpression elimination that the compiler might want to do.

```
type Ref α  =  IORef α
ref a       =  unsafePerformIO (newIORef a)
deref ref   =  unsafePerformIO (readIORef ref)
```

8.1.2 Observable sharing and grammatical categories

We will use references to represent the grammatical categories. The only thing we will use them for is comparison, to detect cycles and recursion in the grammar, so we will never use the function *deref*. Since we are not interested in what the references contain, we could implement the categories as references to nothing in particular, i.e. the unit type.

type Cat = Ref ()

This works fine in the extension, but as noted above, we must turn off the inlining capabilities of the compiler we are using. And in this special case we can do better than that.

If we can guarantee that the categories refer to objects that uniquely determine the categories themselves, we impose even fewer restrictions on the Haskell implementation. Then it will not matter whether the compiler returns a new reference or an already existing reference to an equivalent object. This means that we do not have to turn off inlining and common subexpression elimination.

Since a grammatical category is defined by the set of its productions, we will use references to lists of productions as the type of categories. This is described further in section 8.2.1.

8.2 A grammar-collecting combinator parser

The type of parsers will consist of three things: a category, some way of collecting the grammar, and the semantics which will give the result of the parser.

data CollectingParser s α = P Cat (Collect s) (Semantics s α)

The category will be a unique reference for each occurrence of a parser in the program. The collecting function is an endomorphism that gathers all the categories and context-free productions of the parser and its children. This function will make critical use of the fact that the type of categories has an equality, to be able to stop the infinite recursion that it would otherwise fall into.

type Collect s = Grammar Cat s $\to$ Grammar Cat s

The semantics is a function that transforms a parse tree into a result of the correct type.

type Semantics s α = ParseTree Cat s $\to$ α

These parts of the parser type are defined in the following two subsections. First we define the categories and the grammar collecting function, and then the semantics.

8.2.1 Collecting the grammar

We must be able to create new unique categories while building the grammar, which means that the type of categories must be a reference.

The naive implementation would be to have a reference to nothing in particular, e.g. the unit type. But then smart Haskell implementations might transform all these categories into one single category, as discussed in section 8.1.2. So, we need to come up with something that uniquely determines the category, which the reference can point to. One thing that uniquely determines a category are the production rules that define it. Therefore we say that a category is a reference to a list of rules.[1]

```
newtype Cat  =  Cat (Ref [Production Cat]) deriving Eq
```

Observe that two different categories in a grammar can be defined by the same set of productions, in which case a smart compiler very well could join those two categories into one. Although this will change the grammar, it will not change the language, and it will not change the the structure of the parse trees, and that is enough for our purposes.

To create a new category we create a new reference by giving a list of rules that it can point to.

```
makeCat         ::  [Production Cat] → Cat
makeCat prods   =   Cat (ref prods)
```

Before we start defining the basic combinators, we introduce a helper function *extendGrammar* that creates a new category and computes the collecting function. This helper function takes two arguments, a list of production rules and a list of terminal productions to be added to the grammar and a collecting function that adds more rules to the grammar (which is used when defining the (<⋆>) and (<+>) combinators).

To create the new category, we simply call *makeCat* with the given rules. The collecting function takes a grammar, and the first thing we do is to check if we have already seen the created category. If this is the case, we do nothing at all, since we have already accounted for this category – it is this that stops us from falling into a left- or right-recursive trap. If the category is indeed new, we add the category to the seen categories, add the given productions to the grammar, and call the given collecting function to collect the rest of the grammar.

```
extendGrammar   ::  Ord s ⇒ Set (Production Cat) → (s → Set Cat) →
                               Collect → (Cat, Collect)
extendGrammar prods term collect  =  (cat, collect′)
  where cat                       =  makeCat prods
        collect′ grammar@(cats, start, prods′, term′)
              | cat ‘elemSet‘ cats  =  grammar
              | otherwise           =  collect (unitSet cat <++> cats,
                                                start,
                                                prods <++> prods′,
                                                λs → term s <++> term′ s)
```

[1]We need a **newtype** declaration here, since the type of production rules is defined in terms of the type of categories.

Exercise

This definition can make the terminal function in the grammar very inefficient. Implement this function as a finite map from terminals to sets of categories instead. Which functions have to be changed?

□

Now we are ready to define the parser combinators in terms of the *extendGrammar* function.

The production rule of the *return* parser is a production with an empty right-hand side. There is no need to extend the grammar further so we simply use the *id* function to collect the rest of the grammar. Observe that a smart Haskell implementation can transform the categories of all occurrences of *succeed* into one single category. This will have the effect that the resulting grammar will be smaller, but still recognize the same language, thus making the implementation more efficient.[2]

```
instance Ord s ⇒ PreMonad (CollectingParser s) where
  return a              =  P cat collect sem
    where (cat, collect) =  extendGrammar prods term id
          prods          =  unitSet (cat, [ ])
          term s         =  emptySet
          sem            =  ...
```

For the *sym* parser, one single terminal production has to be added – the production with the given input symbol as the right-hand side.[3]

```
instance Ord s ⇒ Symbol (CollectingParser s) s where
  sym s                 =  P cat collect sem
    where (cat, collect) =  extendGrammar prods term id
          prods          =  emptySet
          term s         =  unitSet cat
          sem            =  ...
```

The *zero* parser has no context-free productions at all, and there is no need to extend the grammar further.[4]

```
instance Ord s ⇒ Monoid (CollectingParser s) where
  zero                  =  P cat collect sem
    where (cat, collect) =  extendGrammar prods term id
          prods          =  emptySet
          term s         =  emptySet
          sem            =  ...
  ...
```

[2]Since this will only have a positive effect, we can do this transformation ourselves by defining *emptyCat* :: Cat on the top-level instead of inside the definition of *return*.

[3]As for the *return* parser, we can join all the categories for each input symbol into one single category, to reduce the size of the grammar.

[4]There is really no need to create a category at all for this parser, we could use *undefined* to reduce the size of the grammar.

There are two productions that have to be added for the choice combinator (<+>), one for each of the two given categories. After we have added the productions, we extend the grammar further by collecting the grammars for the given parsers. Observe that the pattern matching needs to be lazy (by putting a ∼ in front of the patterns), to stop Haskell from evaluating the children parsers before it has created the mother parser.

```
  ...
  ~(P pcat pcollect psem) <+> ~(P qcat qcollect qsem)
                          =  P cat collect sem
    where (cat, collect)  =  extendGrammar prods term (pcollect · qcollect)
          prods           =  makeSet [(cat, [pcat]), (cat, [qcat])]
          term s          =  emptySet
          sem             =  ...
```

And finally for the sequencing (<⋆>), there is one production that has to be added – a sequence of the two given categories. As for (<+>), we need to make the pattern matching lazy on both arguments to avoid falling into infinite recursion.

```
instance Ord s ⇒ Sequence (CollectingParser s) where
  ~(P pcat pcollect psem) <⋆> ~(P qcat qcollect qsem)
                          =  P cat collect sem
    where (cat, collect)  =  extendGrammar prods term (pcollect · qcollect)
          prods           =  unitSet (cat, [pcat, qcat])
          term s          =  emptySet
          sem             =  ...
```

8.2.2 Adding the semantics

We need a way of converting a parse tree to a parse result, so the type of the semantics should be a function from parse trees to results.

```
type Semantics s α  =  ParseTree Cat s → α
```

Calculating the semantic result is straightforward for each of the given combinators. It is only when in the choice combinator we have to check which of the rules the parse tree is applied to.

The three simple parsers have their value already given, so we won't have to look at the parse tree at all. The result of *return a* is the argument *a*.

```
  return a                =  P cat collect sem
    where (cat, collect)  =  ...
          ...
          sem tree        =  a
```

The result of *sym s* is the symbol *s*.

```
sym s                   =  P cat collect sem
  where (cat, collect)  =  ...
        ...
        sem tree        =  s
```

The *zero* parser will never succeed, which means that its semantic function never will be called. So we can use *undefined* as the result.

```
zero                    =  P cat collect sem
  where (cat, collect)  =  ...
        ...
        sem tree        =  undefined
```

For the choice we have to investigate the parse tree, which we know will be a tree with one daughter. If the category of the daughter tree is the first parser's category, we choose that parser's semantic function, and apply it to the daughter tree. Otherwise the category of the daughter tree will be the category of the second parser, so we apply its semantic function to the daughter tree.

```
~(P pcat pcollect psem) <+> ~(P qcat qcollect qsem)
                        =  P cat collect sem
  where (cat, collect)  =  ...
        ...
        sem (_ :^ [tree@(cat' :^ _)])
                        =  (if cat' == pcat then psem else qsem) tree
```

For the sequencing we will get a parse tree with two daughters, and we apply them to their corresponding semantic functions, and then apply the first parser's result to the second's.

```
~(P pcat pcollect psem) <⋆> ~(P qcat qcollect qsem)
                        =  P cat collect sem
  where (cat, collect)  =  ...
        ...
        sem (_ :^ [ptree, qtree])
                        =  (psem ptree) (qsem qtree)
```

Observe that we only partially look at the parse trees. This is because the parse tree reflects the structure of the grammar, so we will always know the structure of the parse tree. It's only in the choice we have to test which branch in the grammar we should continue with.

8.2.3 Parsing

To parse a sentence, we also need an implementation of a parsing algorithm, which can be called *parseCFG*. In the previous chapters there are examples of algorithms that can be used.

First we apply the collecting function to the empty grammar, i.e. empty lists of categories, productions and terminal productions, to get the final productions for the given parser. Then we simply invoke the parsing algorithm with the productions and the starting category, applying the semantic function to each of the resulting parse trees.

```
instance Ord s ⇒ Parser (CollectingParser s) s where
  parseFull (P start collect sem) input
                        =  map sem (parseCFG grammar input)
      where grammar     =  collect emptyGr
            emptyGr     =  (emptySet, start, emptySet, λs → emptySet)
```

Exercise

Implement a parsing function that uses the chart parsing implementation described in chapter 5.

Exercise

Almost all combinator parsers implementing left-recursive grammars are also hidden left-recursive, since it is not type-correct to define $p = p <\star> q$, but instead we have to coerce the result by e.g. $p = \mathit{return}\ f <\star> p <\star> q$. And since *return f* is empty, the category corresponding to p will be hidden left-recursive. Unfortunately our implementation of LR parsing does not work for hidden left-recursive grammars, as explained in section 6.6.

One way to solve this is to remove all empty productions from the grammar. Write a function that does this. It should return a new set of productions, together with a set of empty categories. Use this function to implement LR parsing for the combinators in this chapter.

Exercise

The reader might have noticed that our context-free combinators yields a grammar in almost Chomsky normal form. The *return* parser and the (<+>) combinator break this correspondence. Write a function that transforms out the empty productions from the *return* parser and the unit productions from the (<+>) combinator, to yield a grammar in Chomsky normal form. This grammar can then be used to parse with the CYK parsing algorithm in chapter 7. Then finally the parse trees must be transformed back to the original grammar to be able to apply the semantics function.

□

8.2.4 Other operations on a grammar

There are other things than just parsing we can do with a grammar. We can gather information about the grammar – e.g. if it is left-recursive, cyclic, LR(k) or maybe ambiguous. Another thing is to transform the grammar to make the parsing more efficient, e.g. by removing left-recursiveness.

The method is in all cases the same as the one described, collect the grammar corresponding to the parser structure and call a standard algorithm for the thing you want.

Exercise

Implement a check for left-recursiveness and another for checking if a parser is cyclic.

□

8.3 Discussion

With an implementation of any general parsing algorithm, such as chart parsing described in chapter 5, we can implement a combinator version of a general version of the ambiguous example grammar in section 6.1.3.

$$\begin{array}{lll}
\mathit{expr} & = & \mathit{return\ appOp} <\!\star\!> \mathit{expr} <\!\star\!> \mathit{oper} <\!\star\!> \mathit{expr} \\
 & <\!+\!> & \mathit{return\ toInt} <\!\star\!> \mathit{digit} \\
\textbf{where}\ \mathit{toInt\ c} & = & \mathit{ord\ c} - \mathit{ord}\ \texttt{'0'} \\
\quad \mathit{digit} & = & \mathit{sym}\ \texttt{'0'} <\!+\!> \ldots <\!+\!> \mathit{sym}\ \texttt{'9'} \\
\quad \mathit{appOp\ x\ f\ y} & = & \mathit{f\ x\ y} \\
\quad \mathit{oper} & = & \mathit{return}\ (-) <\!\star\!> \mathit{sym}\ \texttt{'-'} \\
 & <\!+\!> & \mathit{return}\ (+) <\!\star\!> \mathit{sym}\ \texttt{'+'}
\end{array}$$

And this is how it might look like in Hugs.

```
? parseFull expr "1-2-3+4"
[0,6,-8,-2,6]
```

In the rest of this section we discuss some implications and extensions of the ideas introduced in this chapter.

8.3.1 Parametric parsers

Observe that a context-free grammar can only have a *finite* number of categories. This means that one must be careful when using parsers with parameters, or higher-order parsers, as previously discussed in section 2.6.1. The standard example is the *many* combinator, which takes as argument a parser and returns a parser that recognizes a sequence of the argument parser. The naive definition of *many* is not safe, since it leads to non-termination for some Haskell implementations.[5]

$$\mathit{many\ p} \quad = \quad \mathit{return}\ [\,] <\!+\!> p <\!:\!> \mathit{many\ p}$$

[5]Hugs, version Feb 2001, has no problem with this definition. But it falls into non-termination if we instead define $\mathit{many} = \lambda p \rightarrow \ldots$.

The risk is that the call to *many p* on the right-hand side will generate a new category, in which case the grammar collecting function will fall into a non-terminating loop. The correct way to implement *many* is the following, as already done in section 2.8.1.

```
many p      =  ps
  where ps  =  return [ ] <+> p <:> ps
```

It is not always unsafe to use parametric parsers. One has to be careful that the recursion on the parameters terminates, so that the set of introduced categories will be finite.

8.3.2 Adding other basic combinators

It is very common to have a choice of more than two sub-parsers in a grammar. One example is the parser *lowercase* that recognizes lowercase characters.

```
lowercase  =  sym 'a' <+> ... <+> sym 'z'
           =  anyof (map sym [ 'a' .. 'z' ])
           =  foldr (<+>) zero (map sym [ 'a' .. 'z' ])
```

This will introduce $26 \times 2 = 52$ new categories, and $26 \times 3 = 78$ productions, to the grammar, which is a lot more than is necessary. This can make even the fastest parsing algorithm inefficient.

Exercise

Define the *anyof* combinator using the *extendGrammar* function (instead of the default implementation in section 2.5), to introduce fewer new categories and productions to the grammar.

□

Of course one can specialize more combinators to make the grammars even more compact. This is just an example, but an example which will have a great impact on many grammars.

8.3.3 Efficiency

The efficiency of the parsing is of course the same as that of the parsing algorithm used. For example, LR parsing is linear for many unambiguous grammars, and chart parsing is cubic for any context-free grammar.

One problem that is possible to solve is that the references in observable sharing can only be tested for equality, which means that we cannot use sets as ordered lists or binary search trees, as we have done in our implementations in the chapters 5, 6 and 7. The solution to this problem is to translate the categories to e.g. integers before invoking the parsing algorithm.

Exercise

Use the collection of categories returned by the *collect* to write the functions *cat2int* and *int2cat* translating between categories and integers. Then use these functions to transform the productions to use the integer categories instead, to be used in the parsing algorithm. You also need to translate back the integer categories in the resulting parse trees, so that the semantic function can be applied.

□

There is still one problem, which occurs after we have parsed a sentence when we want to calculate the parse results. If the parsed sentence has many solutions, parts of the different parse trees can sometimes be shared, to make the parsing more efficient. Unfortunately, this sharing will be lost when we use the semantic function to calculate the results, since Haskell cannot know that it already has calculated a certain part of a result.

This problem will have an impact if we both have computationally expensive functions in the calculation of the semantics, and a very ambiguous grammar at the same time. One possible way to solve the problem is to use lazy memo functions as in e.g. [7], which is another extension to Haskell.

8.3.4 Conclusion

We have used a small extension to Haskell, observable sharing, to translate a grammar written with parser combinators into a set of context-free productions. Then we can invoke any parsing algorithm, and translate the parse trees back to results of the parser combinators.

The resulting behaviour is similar to parser generators like Happy [28], in that we can make use of static typing and semantic actions at the same time as we can use an efficient parsing algorithm. Advantages over parser generators are that we can make use of Haskell to define new combinators to relieve us from cumbersome grammar writing, and that we don't have to use another formalism than the syntax of standard Haskell. Some additional features of Happy, such as fixity declarations, can be handled by defining a tailor-made combinator that does the work for you. It is also straightforward to implement tests of the resulting grammar – e.g. to test whether it is in a certain class of languages.

But there are disadvantages too. First, we have to use impure and experimental features of the Haskell implementations. In our opinion this is not a severe drawback, since we use the features in a well-behaved way. Second, since we convert the parsers to context-free grammars we can only use context-free combinators. Third, we must be careful when we define parametric parsers, such as the *many* combinator, to avoid falling into non-termination while collecting the grammar. A final disadvantage is that we still have to go through an intermediate parse result, even though it is hidden from the user. This might lead to inefficiences when calculating the semantic actions.

Chapter 9

Final Discussion

In this final chapter we summarize the contents and results from the previous chapters. We discuss the advantages and disadvantages of the different approaches, and in particular we discuss parser combinators vs. parsing algorithms vs. other attempts, such as parser generators. There is a section summarizing the tests that were performed on the parsers and algorithms in the thesis.

The previous chapters have introduced many different combinator parsers and implementations of parsing algorithms, and this final chapter tres to relate them in a bigger perspective.

9.1 A summary of the thesis

The thesis is conceptually divided into two parts, discussing the parsing problem from different perspectives. There is also a final chapter trying to combine the two approaches, to gain the advantages of both.

The first part, chapters 3 and 4, deals with combinator parsers. It is a survey of different possible implementations of parsers; well-known variants as well as brand new ones.

The second part, chapters 5, 6 and 7, deals with parsing algorithms. It consists of implementations of familiar parsing algorithms for general context-free grammars, in a functional setting.

Chapter 2: Grammars and parsers Chapter 2 is an introduction to the basic concepts of the thesis. Context-free grammars are defined, as well as different properties on grammars and sub-classes of grammars; which are also implemented in the functional language Haskell.

The chapter then moves over to parser combinators and defines a Haskell type class hierarchy of parsers. It points out two different versions of parser combinators – context-free and monadic. The monadic combinators are stricly more powerful than the context-free ones in that they can recognize non context-free languages.

9.1.1 Part I: Parser combinators

Chapter 3: Existing parser combinators Chapter 3 starts from the basic type of backtracking combinator parser, which is descended from Wadler's original paper on combintor parsing [44]. The standard continuation transformer and the endomorphism parser are defined to cure some inefficiencies in the original type. This is done by many previous authors, such as Röjemo [36] and Koopman and Plasmeijer [23].

The stack continuation transformer lies hidden within the type of Swierstra's efficient parser [39, 40], and it is extracted in chapter 3. It can make better use of sharing than the standard continuation transformer, but on the other hand it can only implement the context-free parser combinators, not the monadic ones.

The chapter ends by introducing the breadth-first searching stream processor parser, which first appeared in the Fudgets graphics library [3], but not as a parser. It was Claessen [5] who first realized that it is a parser. Both continuation transformers can be applied to the stream parser, yielding a parser which is very efficient for certain kinds of grammars. The parser gets rid of the problem that backtracking parsers cannot discharge previous input until the parse has finally succeeded, which can be a serious space leak for grammars accepting very large input. On the other hand, the idea of breadth-first search is that the stream parser has to remember all possibilities in parallel, which will be a problem for non left-factorized or ambiguous grammars.

Chapter 4: Left-factorizing parser combinators Chapter 4 is devoted to trie structures, or letter trees as they are also called. It is shown that the trie data structure is a parser, doing automatic left-factorization. A problem with tries is that it has to build the structure during parsing. This can be expensive, and more important, it will allocate lots of memory for the trie. The chapter explores the possibilities of sharing of sub-tries, to reduce memory allocation and save time. It turns out that parsers describing regular expressions will be translated to parsers with a call structure resembling finite automata.

It is even possible to associate parse results with regular expressions and still get a finite automata for the parser's call structure. This needs existential quantification in types, which is a simple and well-behaved extension to Haskell.

The chapter ends with describing how to combine the advantages of an efficient backtracking parser with the left-factorization of tries. This final parser transformer is an extraction and simplification of Swierstra's efficient parser [39, 41].

By letting the trie structure return an efficient backtracking parser, instead of a parse result, we can use the backtracking parser on the deterministic parts of the grammar and the trie on the potentially ambiguous parts. The drawbacks are that the transformer can only implement the more limited context-free combinators, and that it introduces extra overhead because of its complexity.

9.1.2 Part II: Parsing algorithms

Chapter 5: Chart parsing Chapter 5 implements a simple and elegant version of bottom-up Kilbury chart parsing [20, 46]. This is one of the many chart parsing variants, which are all based on the data structure of charts. A chart is a set of known facts called edges, which can be depicted as a directed graph over nodes representing positions in the input sequence. The chart parsing process uses inference rules to add new edges to the chart, and parsing is complete when no further edges can be added.

The implementation in this chapter divides the chart into a list of Earley states, named after the top-down Earley parsing algorithm [8]. This makes it possible to parse the input incrementally, building each state in sequence, which in turn gives a clean, elegant and declarative code. The elegance is not traded against efficiency, since the worst-case complexity is shown to be cubic in the length of the input which is as good as any imperative implementation.

Chapter 6: Generalized LR parsing Chapter 6 implements several different versions of generalized LR parsing. LR parsing is a way of pre-compiling the grammar into parse tables, which then can be used to efficiently parse the grammar. Lang [24] showed how to generalize the standard deterministic LR parsing algorithm to be able to parse general context-free grammars, and Tomita [42] implemented a polynomial version of the algorithm.

The most interesting of the implementations in the chapter is an approximation to Tomita's algorithm. The originally complicated graph-structured stack has been simplified to a tree structure, which makes it possible to define an elegant implementation. The graph structure of the stack will still be stored in Haskell memory, because equivalent sub-trees will be shared.

A major drawback is that the algorithm is not polynomial in time but exponential in the length of the input. This is due to the fact that although sub-trees of the stack are shared, it is impossible to know this from within Haskell. This in turn will make the algorithm repeat work unnecessary while traversing the stack.

Chapter 7: CYK parsing Chapter 7 implements an elegant, declarative and purely functional version of the parsing strategy of Cocke, Younger and Kasami [19, 47]. This parsing algorithm is often described as matrix multiplication, but the actual imperative implementations seldom bears any resemblance with the higher-level descriptions. When using a high-level language such as Haskell it is possible retain the idiom, while still having an efficient implementation. The worst-case complexity is cubic in the length of the input, which is as good as one can hope to get.

9.1.3 Tying the parts together

Chapter 8: Grammar-collecting parser combinators Claessen and Sands [6] has developed the mildly impure extension of “observable sharing” which gives the possibility to compare whether two objects are shared or not. This is used in chapter 8 to implement a combinator parser that can collect the grammatical structure in the program. Then this grammar can be used on any of the parsing algorithms described above, and not just recursive descent which is the algorithm implemented by traditional combinator parsers. The parse results from the parsing algorithm are then used to calculate the parse results of the original parser.

There are some disadvantages with this approach, of which the most important is that one must be very careful when defining new combinators. The resulting grammar has to be finite, and vital sharing information must be retained; otherwise the collection procedure might not terminate.

9.2 Different representations of grammars

There are in principle three different ways in which one can define a parser for a given grammar. All three have their own specific advantages and disadvantages; and both the desired implementation and personal taste have influence on which approach is preferred.

We do not give any advice on which approach to use and in which circumstances, but simply describe the three alternatives and let the reader choose which is the method of choice.

Explicit grammar object One approach is to implement the grammar as an explicit object in the host language, and then use a nice parsing algorithm to parse input sentences. This approach is treated in chapters 5, 6 and 7.

The main advantage is that there are no restrictions on the parsing algorithm, which can be tailor-made for our purposes.

The main disadvantage is that we cannot parse directly to the semantic result type we want, but have to go through an intermediate type such as the type of parse trees or charts.

Parser combinators Another version is to implement the grammar directly with parser combinators, and use a suitable parser type. This approach is treated in chapters 3 and 4. Parser combinators form a domain-specific embedded language [14], with all the accompanying features.

The three main advantages are that *i*) we can use the features of the host language to create new combinators; *ii*) the parsers can return the desired semantic action directly instead of using an intermediate type of parse results; and *iii*) the type checker in the host language can catch errors in the grammar, which can be very difficult to find with an untyped grammar. Furthermore, many parser types can be used to implement monadic combinators, which makes it possible to parse non context-free languages.

The main disadvantage with parser combinators is that they all implement the same parsing algorithm – recursive-descent parsing, which is not the most efficient parsing algorithm for many kinds of grammars. This algorithm is also not capable of parsing all possible grammars – left-recursive grammars lead to non-termination, which is a seriuos problem since they are very common in both formal and natural languages.

Parser generator A third alternative is to use a parser generator, such as Happy [28]. A parser generator extends the host language with the ability to define a grammar, together with semantic actions. The grammar file is then compiled to a tailor-made parser in the host language.

This approach combines many of the advantages of the previous two approaches – we can use an efficient parsing algorithm, we can define semantic actions on grammar productions directly, and the semantic results are type checked by the host language.

The main disadvantage is that we cannot use features of the host language to define "macros" for simplifying the grammar writing. Another disadvantage is that we have to learn a new syntax for writing grammar files, but this disadvantage can be considered minor since the syntax often is very natural and simple.

9.3 Summary of test results

Since it is difficult to reason about efficiency issues for lazy functional languages, it is very important to test the implementations on real data.

Every different algorithm and data structure has its own natural method of use. Some are very general and others are tailor-made for a special domain. For combinator parsers and parsing algorithms this means that some parsers are especially efficient on special sub-classes of grammars, and others are more general-purpose.

Most of our testing work has been on combinator parsers. The parsing algorithms are well-known and have been studied for many years. We have also shown complexity results on the implementations in this thesis. Combinator parsers on the other hand have not been studied much – there are not many theoretical results, and the different implementations are not tested against each other.

Recall that the tests reported in this section are quite small. The grammars consist of less than one hundred productions and the test data does not come from real life, except for some natural language data. This means that there are more work to be done to see how the different parsers and algorithms scale up for real life applications.

Finally it can be good to keep in mind that the tested parsers are the ones described in this thesis. The continuation based pairing trie parser consists of two parser transformers applied to a base parser, and each transformer introduces some extra overhead which can be optimized. I.e. some of the parsers are possible to optimize, which has not been done here. And if we are only interested in e.g. deterministic grammars there are sometimes further possible optimizations to be done.

9.3.1 Sub-classes of grammars

We can divide the spectrum of context-free grammars in several different ways, and here we have chosen four different dimensions on which to categorize grammars.

Complexity We say that a grammar is simple when its language is a regular language. A complex grammar is the opposite, e.g. an expression grammar with nested parentheses. We can also talk about complexity of the input – e.g. how deeply nested the input is.

Left-factorization Since some of the parsers do left-factorization automatically, it can be interesting to compare ordinary grammars with already left-factorized grammars.

Ambiguity A grammar can finally be ambiguous or not. This distinction often coincides with the distinction between natural and formal languages – a natural language is ambiguous and a formal language is unambiguous.

9.3.2 Testing combinator parsers

From the different dimensions described above we created seven test grammars, and on each of the grammars we tested a number of different inputs. In appendix

B the exact figures of the tests can be found. The seven test grammars were all context-free, to be parseable by every combinator parser.

- Two very simple grammars for recognizing sequences of numbers – one of them is left-factorized and the other is not.[1] The two grammars were tested on randomly generated text files consisting of long lines of numbers.

- Two more complex grammars recognizing nested mathematical expressions – one of them left-factorized in the same way as above. The test data consisted of randomly generated huge mathematical expressions, either in English or in standard syntax. The parsers were tested on flat expressions as well as very nested expressions.

- Three ambiguous natural language grammars recognizing Swedish declarative sentences. One is left-factorized, one is not left-factorized, and the third is a quite complex, partly left-factorized, natural language grammar. The simple grammars were tested on randomly generated sentences, of which many were quite ambiguous. The complex grammar was tested on sentences from a real Swedish corpus. Only sentences recognized by the grammar were used in the test data.

In this section we give two tables summarizing the test results for the different grammars. The first of the tables contains the unambiguous grammars and the second contains the ambiguous grammars. Observe that the last line in the tables, is the PairTrie transformer applied to the most efficient of the standard parsers – we do not list all possible variants for the PairTrie to make the table manageable. We do not list the standard trie parser, since it is only efficient for recognition as mentioned in section 4.3. We also leave out the ambiguous trie for the unambiguous grammars, and the ordinary trie for the ambiguous grammars for obvious reasons.

In appendix B the results for all tested parsers can be found. Observe that the results can be improved for some of the parsers – the applicatations of parser transformers can be hard-coded into a tailor-made parser, thus reducing the overhead introduced by having transformers. Observe also that the stack continuation transformer and the pairing trie transformer both prohibit the use of monadic combinators. So, if you want to use monadic parsing, these parsers are out of the question.

There are four possible symbols in the tables; minus, zero, plus and double-plus. There are no absolute meaning assigned to these symbols, just a relative comparision between the parsers for one specific grammar. I.e. a double-plus means that the parser is "the best" parser for that specific grammar, and a minus means that the parser is very slow compared to the other parsers for the grammar.

[1]To be more specific, the first grammar recognizes numbers written with digits, and the second recognizes numbers written in plain English.

Memory usage The symbols in the tables only regard the time for the parsing to finish – memory usage cannot be read from the tables. Thus it is interesting to know how much memory the parsers allocate while parsing. According to the tests performed, this is quite straightforward and not at all unnatural.

The trie parsers are eating up much memory almost always, and the backtracking parsers are quite memory efficient. The pairing trie parsers lies somewhere in between the two extremes. The stream parser finally is very memory efficient on the determinstic grammars – better than the backtracking parsers – but extremely memory consuming on the ambiguous grammars – much worse than the trie parsers.

Unambiguous grammars As general parsers for unambiguous grammars, the endomorphism continuation parser, the stream continuation parser and the pairing trie parser are all well suited. The trie parser is efficient for very simple non left-factorized grammars, which means that it is suited for tokenization/lexing or morphological analysis. The continuation transformers all work well on unambiguous grammars, with the stack continuation transformer slightly better than the ordinary continuation transformer. The efficiency of the stream parser decreases somewhat when the grammar is not left-factorized.

Unambiguous		**Num**	**Num (LF)**	**Expr**	**Expr (LF)**
Standard	(3.2)	–	–	0	0
StdCont	(3.3.2)	–	–	0	0
StdStack	(3.4.2)	–	–	0	0
StdEndoCont	(3.3.4)	0	+	+	+
StdEndoStack	(3.4.2)	0	++	+	+
Stream	(3.5)	–	–	+	0
StreamCont	(3.5.3)	0	+	++	++
StreamStack	(3.5.4)	0	++	++	++
ExTrie	(4.3.3)	++	0	–	–
PairTrie	(4.4.2)	+	0	++	0

Ambiguous grammars For ambiguous grammars, the ambiguous trie parser stands out as being much more efficient than any of the others. On second place comes the pairing trie parser, thus making it a good general-purpose parser for both deterministic and ambiguous grammars. The stream parsers work especially poorly on ambiguous grammars.

Observe that we calculated *all* possible parse results and not only the first, or the first ten. If we were interested in only a few of the results, the results for the standard parsers would probably be much better.

Ambiguous		**NL**	**NL (LF)**	**NL (Complex)**
Standard	(3.2)	–	–	0
StdCont	(3.3.2)	–	0	0
StdStack	(3.4.2)	–	+	0
StdEndoCont	(3.3.4)	–	0	0
StdEndoStack	(3.4.2)	–	+	0
Stream	(3.5)	–	–	0
StreamCont	(3.5.3)	–	–	–
StreamStack	(3.5.4)	–	–	–
AmbExTrie	(4.3.3)	++	++	++
PairTrie	(4.4.2)	+	+	–

9.3.3 Testing parsing algorithms

The three different parsing algorithms described in chapters 5, 6 and 7 was tested on three different natural language grammars.

- The first grammar is a trivial grammar with only 5 productions, for testing very ambiguous data. It was tested on ambiguous sentences of increasing length.
- The second grammar is a simple natural language grammar, with 19 productions. This grammar is similar to the left-factorized version of the natural language parser; with some differences stemming from the fact that parsers cannot handle left-recursion. It was tested on real data from a Swedish corpus of newspaper texts, of which the grammar recognized about 15% of the sentences.
- The third grammar is a slightly more complex natural language grammar consisting of 31 productions, similar to the complex natural language parser. This grammar was tested on the same corpus as the simple grammar, and this time it recognized about 40% of the sentences.

The last two grammars were also transformed to Chomsky normal form, to be able to test the CYK parsing algorithm.

The testing procedure calculated all possible parse trees, and it can of course be discussed if this is the best method. Another possible variant is to calculate the parse graph or a chart, which is known to be polynomial in space, even when there are an exponential number of parse trees. The latter is often a better approach in large-scale natural language applications, where they often have thousands of possible parse trees of which only some are chosen for the semantic processing.

Memory behaviour The memory behaviour is similar for all the different implementations, which should come as no surprise since their space complexity is only quadratic in the length of the input. The exponential number of parse trees did not have an impact, probably because the trees were calculated one at the time and then discarded.

Efficiency The CYK algorithm was faster than chart parsing, probably because it is a simpler algorithm which doesn't have to calculate minimal fix-point sets.

The Tomita approximation was the fastest of the three LR implementations. It was more efficient than chart parsing and CYK parsing on the simple input without many ambiguities.

On very ambiguous input the Tomita implementation became terribly slow, which probably is because the implementation is just an approximation with an exponential worst-case complexity.

If we had been interested in only the first few parse trees, the depth-first LR implementation would probably be the most efficient one.

9.4 Future work

The area of parsing is wide and there are many possible paths leading out of this thesis. Since there has been so little work done on parsing in a functional framework, it is in principle possible to continue with anything parsing related. In this final section we will just hint at some areas where there are interesting questions to be answered.

Parser combinators Much research has been conducted to implement different combinator parsers, but almost no-one have studied combinators more theoretically.

One very interesting issue is to explore the expressiveness of different parser combinators. How will the language be affected if we introduce a special combinator, such as a pruning combinator? Or, what languages can be recognized if we only accept a limited range of functions as the second argument to the monadic bind? A related question is how to make parser combinators safer – how can we check that a parser is guaranteed to terminate?

Grammar formalisms It is interesting to look at how more expressive grammar formalisms than context-free grammars can be elegantly implemented in a pure functional language.

Parsing algorithms One very burning question from this thesis is how to implement a real Tomita LR parser in a functional way. The imperative algorithm is both big, hard to read, non-intuitive and... well... imperative. Is it possible to implement an efficient Tomita parser in an elegant and declarative way? Attemps at this have already been conducted by Callaghan and Medlock [29].

Otherwise one can always implement other parsing algorithms, such as left-corner or head-corner chart parsing; or even parsing algorithms for other grammar formalisms. Remember that the goal should be to implement the algorithm in an elegant and declarative way without losing the efficiency of the imperative algorithm.

Parser generators One interesting issue is if it is possible to combine more of the features of parser generators into the grammar-collecting parser of chapter 8. Maybe also the other way around – is it possible to use something from chapter 8 when designing a parser generator?

Evaluation The evaluation in this chapter is not very thorough, and a more comprehensive evaluation would be nice. The parsers and algorithms should be tested on real-sized applications, and in a more systematic way. Especially it would be interesting with a comparision between combinator parsers, parsing algorithms and parser generators. Then one might draw more interesting conclusions about the usefulness of different approaches.

Another interesting evaluation would be to try to compare the different approaches with each other on other aspects, such as how error prone they are and how easy they are to use. Of course such an evaluation would be very difficult, since these things often boil down to personal taste.

Appendix A

Sets and Finite Maps

In this appendix we give example implementations of sets and finite maps, the former as ordered lists and the latter as ordered association lists.

A.1 Sets as ordered lists

One of the obvious representations of sets is as ordered lists. This means that the type Set α is just a list of α:s. t

$$\textbf{type}\ \mathsf{Set}\ \alpha \quad = \quad [\,\alpha\,]$$

The empty set is the empty list and the singleton set is the singleton list

$$\begin{array}{lcl} \mathit{emptySet} & :: & \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Set}\ \alpha \\ \mathit{emptySet} & = & [\,] \\[1ex] \mathit{unitSet} & :: & \mathsf{Ord}\ \alpha \Rightarrow \alpha \to \mathsf{Set}\ \alpha \\ \mathit{unitSet}\ x & = & [\,x\,] \end{array}$$

The test for emptiness is already defined in the *null* predicate, and membership in the *elem* predicate.

$$\begin{array}{lcl} \mathit{isEmpty} & :: & \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Set}\ \alpha \to \mathsf{Bool} \\ \mathit{isEmpty} & = & \mathit{null} \\[1ex] \mathit{elemSet} & :: & \mathsf{Ord}\ \alpha \Rightarrow \alpha \to \mathsf{Set}\ \alpha \to \mathsf{Bool} \\ \mathit{elemSet} & = & \mathit{elem} \end{array}$$

We can take the union of two sets, defined as the function *plus*. This is a standard merge, as in the merge sort algorithm.

```
(<++>)        ::  Ord α ⇒ Set α → Set α → Set α
[ ] <++> ys    =  ys
xs <++> [ ]    =  xs
xs@(x : xs′) <++> ys@(y : ys′)
               =  case compare x y of
                    LT → x : (xs′ <++> ys)
                    GT → y : (xs <++> ys′)
                    EQ → x : (xs′ <++> ys′)
```

The naive way of doing the union of a list of sets is to fold the list with (<++>), but it is more efficient to split the list of sets into two, union then separately and then adding the results together. This is really a version of merge sort.

```
union                    ::  Ord α ⇒ [Set α] → Set α
union [ ]                 =  emptySet
union [xs]                =  xs
union xyss                =  union xss <++> union yss
  where (xss, yss)        =  split xyss
        split [ ]         =  ([ ], [ ])
        split [x]         =  ([x], [ ])
        split (x : y : xys) = let (xs, ys) = split xys in (x : xs, y : ys)
```

To turn an unordered list into a set, we turn each element of the list into a singleton set, and then apply *union*.

```
makeSet      ::  Ord α ⇒ [α] → Set α
makeSet xs    =  union (map unitSet xs)
```

Getting the list of elements in a set is just the identity function, as well as creating a set from an ordered list.

```
elems   ::  Ord α ⇒ Set α → [α]
elems    =  id

ordSet  ::  Ord α ⇒ [α] → Set α
ordSet   =  id
```

As the definition for the *limit* function we use the one given in section 2.2.1.

```
limit                    ::  Ord α ⇒ (α → Set α) → Set α → Set α
limit more start          =  limit′ start start
  where limit′ old new
          | isEmpty new′  =  old
          | otherwise     =  limit′ (new′ <++> old) (new′ <--> old)
        where new′        =  union (map more new)
```

The set difference used in the *limit* function can be calculated in the same manner as the set union.

```
(<−>)          ::  Ord α ⇒ Set α → Set α → Set α
[ ] <−> ys     =   [ ]
xs <−> [ ]     =   xs
xs@(x : xs′) <−> ys@(y : ys′)
               =   case compare x y of
                     LT → x : (xs′ <−> ys)
                     GT → xs <−> ys′
                     EQ → xs′ <−> ys′
```

A.2 Finite maps as ordered association lists

We can extend the representation of sets as ordered lists to represent finite maps as ordered association lists. An association list is a list of pairs of key and values. The list is ordered on the keys, and there are no duplicate keys in the list.

```
type Map s α  =  [(s, α)]
```

The empty map is represented as the empty list and to map a symbol s to a value a we use a singleton list.

```
emptyMap  ::  Ord s ⇒ Map s α
emptyMap  =   [ ]

(|→)      ::  Ord s ⇒ s → α → Map s α
s |→ a    =   [(s, a)]
```

The test for emptiness is trivial, and to lookup a value in the map we can use the standard *lookup* function.

```
isEmptyMap  ::  Ord s ⇒ Map s α → Bool
isEmptyMap  =   null

(?)         ::  Ord s ⇒ s → Map s α → Maybe α
s ? ass     =   lookup ass s
```

To merge two association lists we need to know what to do with two conflicting values for the same input symbol. Otherwise it is a standard merge, like the one used in the merge sort algorithm.

```
mergeWith   :: Ord s ⇒ (α → α → α) → Map s α → Map s α → Map s α
mergeWith join [ ] bss  =  bss
mergeWith join ass [ ]  =  ass
mergeWith join ass@(x@(s, a) : ass′) bss@(y@(t, b) : bss′)
                        =  case compare s t of
                             LT → x : mergeWith join as′ bs
                             GT → y : mergeWith join as bs′
                             EQ → (s, join a b) : mergeWith join as′ bs′
```

To create a finite map from a list of key-value pairs, we extend the union function on ordered lists to association lists

```
makeMapWith   :: Ord s ⇒ (α → α → α) → [ (s, α) ] → Map s α
makeMapWith join [ ]         =  emptyMap
makeMapWith join [ (s, a) ]  =  s |→ a
makeMapWith join xyss        =  mergeWith join (makeMapWith join xss)
                                                (makeMapWith join yss)
  where (xss, yss)           =  split xyss
        split [ ]            =  ([ ], [ ])
        split [ x ]          =  ([ x ], [ ])
        split (x : y : xys)  =  let (xs, ys) = split xys in (x : xs, y : ys)
```

The functions to return the association list of a finite map and to create a finite map from an ordered association list are just the identity functions.

```
assocs  :: Ord s ⇒ Map s α → [ (s, α) ]
assocs  =  id

ordMap  :: Ord s ⇒ [ (s, α) ] → Map s α
ordMap  =  id
```

Finally, we can map a function over the values of a finite map.

```
mapMap        :: Ord s ⇒ (α → β) → Map s α → Map s β
mapMap f ass  =  [ (s, f a) | (s, a) ← ass ]
```

Appendix B

Test Results

In this appendix we list the results of the performed tests which are summarized in chapter 9. Each parser and algorithm have been tested on three or four different test files of different sizes and complexity. In the tables are listed the total time to finish the task, together with a number indicating how many percent of that time was spent on garbage collection. The third figure is the maximum amount of allocated memory during the process. We have left out the total amount and the average amount of allocated memory.

The tests were performed on a Sun Ultra 80 with 4GB main memory. The compiler was GHC version 5.02.1, and all files were compiled with optimization turned on. The tests were then run with a maximum heap size of 1GB and a maximum stack size of 10MB.

The timings are not quite exact, since each test varied a bit for each run. But they are within ±5% of the given value. The allocated memory is exact though.

B.1 Combinator parsers

The Number parser The test data consisted of a number of long lines of numbers written in plain English. The length of the lines veried from 2000 to 65000 characters, or 100 to 3000 numbers. The first file consisted of many short lines, and then the line length increased until the last file which consisted of a few very long lines.

$$\begin{array}{lcl}
\mathit{numbers} & = & \mathit{fmap}\ \mathit{sum}\ (\mathit{number} <:> \mathit{many}\ (\mathit{sym}\ \texttt{'␣'} \star\!> \mathit{number})) \\
\mathit{number} & = & \mathit{sub}_{1000}\ <\!\!+\!\!>\ \mathit{sup}_{1000}\ <\!\!+\!\!>\ \mathit{fmap}\ (+)\ \mathit{sup}_{1000} <\!\star\!> \mathit{sub}_{1000} \\
\\
\mathit{sup}_{1000} & = & \mathit{fmap}\ (\mathit{const} \cdot (1000*))\ \mathit{sub}_{1000} <\!\star\!> \mathit{thousand} \\
\mathit{sub}_{1000} & = & \mathit{sub}_{100}\ <\!\!+\!\!>\ \mathit{sup}_{100}\ <\!\!+\!\!>\ \mathit{fmap}\ (+)\ \mathit{sup}_{100} <\!\star\!> \mathit{sub}_{100} \\
\\
\mathit{sup}_{100} & = & \mathit{fmap}\ (\mathit{const} \cdot (100*))\ \mathit{digit} <\!\star\!> \mathit{hundred} \\
\mathit{sub}_{100} & = & \mathit{digit}\ <\!\!+\!\!>\ \mathit{teen}\ <\!\!+\!\!>\ \mathit{ten}\ <\!\!+\!\!>\ \mathit{fmap}\ (+)\ \mathit{ten} <\!\star\!> \mathit{digit} \\
\\
\mathit{thousand} & = & \mathit{syms}_0\ \texttt{"thousand"} \\
\mathit{hundred} & = & \mathit{syms}_0\ \texttt{"hundred"} \\
\\
\mathit{ten} & = & \mathit{syms}_0\ \texttt{"twenty"} \star\!> \mathit{return}\ 20 \quad <\!\!+\!\!> \quad \ldots \\
 & <\!\!+\!\!> & \mathit{syms}_0\ \texttt{"ninety"} \star\!> \mathit{return}\ 90 \\
\\
\mathit{teen} & = & \mathit{syms}_0\ \texttt{"ten"} \star\!> \mathit{return}\ 10 \quad <\!\!+\!\!> \quad \ldots \\
 & <\!\!+\!\!> & \mathit{syms}_0\ \texttt{"nineteen"} \star\!> \mathit{return}\ 19 \\
\\
\mathit{digit} & = & \mathit{syms}_0\ \texttt{"one"} \star\!> \mathit{return}\ 1 \quad <\!\!+\!\!> \quad \ldots \\
 & <\!\!+\!\!> & \mathit{syms}_0\ \texttt{"nine"} \star\!> \mathit{return}\ 9
\end{array}$$

Number	**Test 1**	**Test 2**	**Test 3**
Standard	20.4s (14%) 0.5M	27.3s (17%) 2.5M	> 1000s
StdCont	14.8s (20%) 0.2M	59.7s (43%) 1.0M	> 1000s
StdStack	15.6s (21%) 0.3M	58.4s (40%) 1.2M	> 1000s
StdEndoCont	10.3s (20%) 0.2M	11.6s (21%) 0.7M	56.8s (27%) 4.2M
StdEndoStack	11.3s (23%) 0.2M	12.1s (29%) 0.9M	40.2s (35%) 4.9M
Stream	14.2s (5%) 0.1M	24.2s (6%) 0.1M	318.4s (21%) 3.7M
StreamCont	7.5s (7%) 0.1M	8.2s (6%) 0.4M	40.4s (13%) 2.3M
StreamStack	8.9s (9%) 0.1M	8.7s (8%) 0.5M	27.4s (12%) 3.3M
ExTrie	1.8s (47%) 4.3M	1.8s (49%) 4.9M	4.1s (44%) 8.8M
AmbExTrie	1.8s (47%) 4.5M	1.9s (48%) 4.8M	4.6s (52%) 11.6M
PairTrieStd	9.3s (38%) 3.6M	18.0s (28%) 5.5M	683.5s (70%) 21.8M
PairTrieStdCont	4.8s (29%) 3.1M	10.5s (24%) 3.4M	316.8s (49%) 10.3M
PairTrieStdStack	6.1s (36%) 3.2M	10.4s (28%) 3.3M	205.4s (37%) 11.8M
PairTrieStdEndoCont	3.9s (35%) 3.2M	4.8s (32%) 3.3M	31.6s (21%) 9.3M
PairTrieStdEndoStack	5.1s (39%) 3.2M	5.0s (46%) 4.0M	14.2s (43%) 11.5M

As can be seen, the trie parsers are the clear winners, but they also allocate the most memory. Otherwise one can see that the continuation transformers give a real improvement, and the stack continuation more then the ordinary. The pairing trie transformer also give an improvement, but it will instead use up memory.

The left-factorized Number parser The test data consisted of long lines of numbers written as sequences of digits. The length of the lines varied between 1200 and 12500 characters, or between 250 and 2500 numbers. The first file consisted of many short lines, and then the line length increased until the last file which consisted of a few very long lines.

```
numbers          =  fmap sum (number <:> many (sym '␣' ⋆> number))

number           =  fmap conv (digit <:> many digit)
  where conv     =  foldl (λn d → n ∗ 10 + d) 0

digit            =  fmap conv (sat isDigit)
  where conv c   =  toInteger (ord c − ord '0')
```

Number (LF)	**Test 1**	**Test 2**	**Test 3**
Standard	10.7s (17%) 0.3M	18.1s (21%) 1.0M	130.8s (36%) 3.3M
StdCont	4.6s (9%) 0.1M	8.0s (16%) 0.1M	55.7s (35%) 1.1M
StdStack	3.5s (18%) 0.2M	5.4s (13%) 0.4M	33.3s (24%) 1.3M
StdEndoCont	2.3s (13%) 0.1M	2.8s (10%) 0.1M	13.5s (15%) 0.9M
StdEndoStack	1.2s (31%) 0.1M	0.8s (41%) 0.3M	1.6s (45%) 1.2M
Stream	6.7s (8%) 0.0M	11.2s (13%) 0.3M	72.4s (27%) 0.9M
StreamCont	1.9s (13%) 0.1M	2.4s (11%) 0.0M	12.2s (15%) 0.4M
StreamStack	0.9s (30%) 0.1M	0.6s (40%) 0.2M	1.2s (42%) 0.7M
ExTrie	6.5s (56%) 22.1M	5.2s (57%) 20.3M	6.4s (55%) 22.1M
AmbExTrie	6.6s (53%) 22.1M	5.5s (59%) 22.1M	6.5s (55%) 22.1M
PairTrieStd	17.6s (23%) 20.3M	29.6s (26%) 22.1M	307.6s (60%) 22.1M
PairTrieStdCont	24.6s (25%) 20.3M	53.8s (36%) 22.1M	866.2s (71%) 20.3M
PairTrieStdStack	22.8s (24%) 22.1M	47.0s (35%) 20.3M	843.7s (66%) 22.1M
PairTrieStdEndoCont	7.2s (41%) 22.1M	7.1s (37%) 22.1M	18.8s (25%) 20.3M
PairTrieStdEndoStack	6.3s (44%) 22.1M	4.9s (48%) 22.1M	6.7s (49%) 20.3M

Now the tries and pairing tries are no longer the fastest parsers, but this is not because they have become slower on this input, but rather that the stream and backtracking parsers are especially well suited for the purpose. The stack continuation give an exceptionally good improvement.

The Expression parser The test data consisted of mathematical expressions written in plain English. The length of the lines varied between 300 and 23500 characters, or between 30 and 2300 words. The first of the test files consisted of very simple (non-nested) expressions, and then the complexity of the input increased until the last of the files with deeply nested expressions.

```
expr      =    fmap apply expr0 <⋆> many (fmap flip oper <⋆> expr0)
  where apply = foldl (λx f → f x)

expr0     =    fmap const (listoper <⋆> exprs) <⋆> sym ';'
         <+>   number
         <+>   sym '(' ⋆> fmap const expr <⋆> sym ')'
         <+>   syms0 "minus␣" ⋆> fmap negate expr0

oper      =    syms0 "␣plus␣" ⋆> return (+)
         <+>   syms0 "␣minus␣" ⋆> return (−)
         <+>   syms0 "␣times␣" ⋆> return (∗)
         <+>   syms0 "␣divided␣by␣" ⋆> return div

exprs     =    fmap addlast exprs0 <⋆> (syms0 "␣and␣" ⋆> expr)
  where addlast ns n = ns ++ [ n ]

exprs0    =    expr <:> many (syms0 ",␣" ⋆> expr)

listoper  =    syms0 "the␣sum␣of␣" ⋆> return sum
         <+>   syms0 "the␣product␣of␣" ⋆> return product
```

The *number* parser is the same parser as defined on page 132.

Expression	Test 1	Test 2	Test 3
Standard	12.2s (19%) 2.5M	9.7s (19%) 1.8M	8.7s (18%) 2.3M
StdCont	26.4s (41%) 1.0M	5.8s (35%) 0.9M	5.0s (31%) 1.0M
StdStack	26.3s (39%) 1.2M	6.6s (37%) 1.2M	5.7s (33%) 1.3M
StdEndoCont	6.1s (25%) 0.6M	5.8s (26%) 0.6M	5.4s (28%) 0.6M
StdEndoStack	6.1s (28%) 0.9M	5.7s (28%) 0.8M	5.6s (26%) 0.9M
Stream	9.9s (7%) 0.6M	5.9s (6%) 0.3M	5.6s (8%) 0.6M
StreamCont	4.1s (7%) 0.3M	3.6s (4%) 0.1M	3.4s (5%) 0.2M
StreamStack	4.2s (12%) 0.5M	4.0s (11%) 0.4M	3.9s (11%) 0.6M
ExTrie	1.2s (54%) 3.7M	6.8s (69%) 32.1M	12.4s (73%) 69.0M
AmbExTrie	1.2s (47%) 3.6M	6.8s (64%) 29.2M	12.6s (70%) 64.5M
PairTrieStd	7.5s (38%) 3.9M	4.3s (49%) 4.1M	4.0s (50%) 6.1M
PairTrieStdCont	10.6s (33%) 2.6M	2.7s (51%) 2.4M	2.5s (44%) 3.6M
PairTrieStdStack	10.9s (34%) 2.9M	3.6s (47%) 2.8M	3.1s (48%) 4.4M
PairTrieStdEndoCont	2.6s (37%) 2.3M	2.3s (37%) 2.8M	2.1s (52%) 3.2M
PairTrieStdEndoStack	3.1s (45%) 2.4M	3.1s (42%) 2.5M	3.0s (47%) 3.9M

The winner this time is the pairing trie, followed by the stream parser. Observe that this time the standard continuation transformer has the best performance, even if just by a little. Interesting is the behaviour of the trie parsers – on the simple input they are very efficient indeed, but as soon as we introduce some nesting, they become slower and slower. Also notice that the memory allocation for the tries is tightly connected with the running time.

The left-factorized Expression parser The test data consisted of mathematical expressions written in standard Haskell syntax. The length of the lines varied between 200 and 20000 characters, or between 20 and 9000 tokens. The first of the test files consisted of very simple (non-nested) expressions, and then the complexity of the input increased until the last of the files with deeply nested expressions.

```
expr     =   fmap apply expr0 <*> many (fmap flip oper <*> expr0)
  where apply = foldl (λx f → f x)

expr0    =   listoper <*> (sym '[' *> fmap const exprs <*> sym ']')
        <+>  number
        <+>  sym '(' *> fmap const expr <*> sym ')'
        <+>  syms0 "-" *> fmap negate expr0

oper     =   syms0 "+" *> return (+)
        <+>  syms0 "-" *> return (−)
        <+>  syms0 "*" *> return (∗)
        <+>  syms0 "/" *> return div

exprs    =   expr <:> many (sym ',' *> expr)

listoper =   syms0 "sum␣" *> return sum
        <+>  syms0 "product␣" *> return product
```

The *number* parser is the left-factorized version defined on page 133.

Expression (LF)	Test 1	Test 2	Test 3
Standard	48.3s (30%) 2.1M	11.3s (14%) 1.6M	5.8s (22%) 2.3M
StdCont	112.0s (49%) 1.3M	14.1s (16%) 0.3M	3.0s (33%) 1.0M
StdStack	91.4s (42%) 1.4M	12.5s (16%) 0.8M	2.9s (33%) 1.0M
StdEndoCont	7.0s (18%) 0.9M	2.8s (12%) 0.3M	2.4s (17%) 0.7M
StdEndoStack	2.6s (37%) 1.1M	1.9s (23%) 0.7M	2.5s (22%) 1.0M
Stream	29.2s (20%) 0.5M	6.3s (9%) 0.2M	3.3s (10%) 0.5M
StreamCont	5.7s (12%) 0.4M	2.3s (13%) 0.3M	1.8s (16%) 0.4M
StreamStack	1.7s (27%) 0.5M	1.4s (24%) 0.4M	1.7s (26%) 0.5M
ExTrie	4.4s (55%) 22.1M	9.0s (58%) 43.7M	25.1s (68%) 112.6M
AmbExTrie	4.5s (56%) 22.1M	9.0s (58%) 43.7M	23.6s (67%) 100.1M
PairTrieStd	75.7s (37%) 30.4M	16.6s (24%) 22.1M	8.8s (37%) 22.1M
PairTrieStdCont	131.8s (54%) 31.4M	108.9s (40%) 31.2M	10.0s (52%) 22.1M
PairTrieStdStack	99.6s (45%) 31.5M	94.3s (41%) 31.3M	9.7s (53%) 20.3M
PairTrieStdEndoCont	11.3s (35%) 30.0M	6.4s (36%) 20.3M	6.5s (41%) 22.1M
PairTrieStdEndoStack	6.9s (56%) 31.0M	6.2s (53%) 31.3M	6.4s (46%) 22.1M

When the grammar is left-factorized the stream parser wins, followed by the backtracking parser. The stack continuation is much better than the standard continuation on simple input, but there is not difference on the nested input. The tries and the pairing tries lose this time, furthermore they allocate lots of memory. One interesting thing is that the standard parser and the stream parser (without any continuations) are very slow on the simple input, but become more efficient on nested data.

The NL parser The natural language parser is a quite simple, but highly ambiguous, natural language grammar for Swedish, with a lexicon consisting of 960 words. The tokenization and morphological analysis (word class tagging) were all part of the grammar. The test data consisted of seven lines each of between 90 and 240 characters, or between 10 and 30 word sentences. The ambiguities varied between 2 and 700 different results for a sentence.

$$\begin{array}{lcl}
\mathit{sentences} & = & \mathit{sentence} <:> \mathit{many\ sentence} \\
\mathit{sentence} & = & \mathit{mapNode2}\ \mathsf{S}\ \mathit{np} <\star> \mathit{vp} \\
\mathit{vp} & = & \mathit{fmap}\ (\mathsf{VP}{:}\hat{})\ (\mathit{vp0} <:> \mathit{many\ pp}) \\
\mathit{vp0} & = & \mathit{mapNode1}\ \mathsf{VP}\ \mathit{verb} \\
 & <\!+\!> & \mathit{mapNode2}\ \mathsf{VP}\ \mathit{verb} <\star> \mathit{np} \\
\mathit{np} & = & \mathit{fmap}\ (\mathsf{NP}{:}\hat{})\ (\mathit{np0} <:> \mathit{many}\ (\mathit{pp} <\!+\!> \mathit{rp})) \\
\mathit{np0} & = & \mathit{mapNode1}\ \mathsf{NP}\ (\mathit{name}\ <\!+\!>\ \mathit{pronoun}\ <\!+\!> \mathit{noun}) \\
 & <\!+\!> & \mathit{mapNode2}\ \mathsf{NP}\ \mathit{ap} <\star> \mathit{noun} \\
 & <\!+\!> & \mathit{mapNode2}\ \mathsf{NP}\ \mathit{det} <\star> \mathit{noun} \\
 & <\!+\!> & \mathit{mapNode3}\ \mathsf{NP}\ \mathit{det} <\star> \mathit{ap} <\star> \mathit{noun} \\
\mathit{pp} & = & \mathit{mapNode2}\ \mathsf{PP}\ \mathit{prep} <\star> \mathit{np} \\
\mathit{rp} & = & \mathit{mapNode2}\ \mathsf{RP}\ \mathit{relpro} <\star> \mathit{vp} \\
\mathit{ap} & = & \mathit{fmap}\ (\mathsf{AP}{:}\hat{})\ (\mathit{adj} <:> \mathit{many\ adj}) \\
 & & \\
\mathit{adj} & = & \mathit{leaf}\ [\,\texttt{"allvarlig"},\ \texttt{"gamle"},\ \texttt{"tekniskt"},\ \ldots] \\
\ldots & & \\
\mathit{verb} & = & \mathit{leaf}\ [\,\texttt{"arbetar"},\ \texttt{"omfattar"},\ \texttt{"pekade"},\ \ldots] \\
 & & \\
\mathit{mapNode1}\ n & = & \mathit{fmap}\ (\lambda a \to n :\hat{}\ [\,a\,]) \\
\mathit{mapNode2}\ n & = & \mathit{fmap}\ (\lambda a\ b \to n :\hat{}\ [\,a,\ b\,]) \\
\mathit{mapNode3}\ n & = & \mathit{fmap}\ (\lambda a\ b\ c \to n :\hat{}\ [\,a,\ b,\ c\,]) \\
\mathit{leaf\ strs} & = & \mathit{anyof}\ [\,\mathit{syms}_0\ \mathit{str} \star\!> \mathit{return}\ (\mathsf{Leaf}\ \mathit{str})\ \mid\ \mathit{str} \leftarrow \mathit{strs}\,]
\end{array}$$

NLParser	**Test 1**	**Test 2**	**Test 3**
Standard	7.1s (3%) 0.4M	163.1s (3%) 1.2M	376.8s (3%) 1.4M
StdCont	5.4s (30%) 0.4M	130.8s (35%) 0.6M	305.1s (36%) 0.7M
StdStack	4.3s (12%) 0.4M	96.7s (13%) 1.0M	222.7s (13%) 1.2M
StdEndoCont	6.6s (34%) 0.4M	160.8s (36%) 0.6M	373.1s (37%) 0.7M
StdEndoStack	9.3s (19%) 0.4M	218.2s (23%) 1.1M	495.8s (24%) 1.2M
Stream	6.0s (28%) 1.8M	167.1s (39%) 43.6M	420.7s (41%) 88.1M
StreamCont	6.8s (52%) 2.9M	212.7s (61%) 79.1M	514.1s (62%) 178.0M
StreamStack	8.7s (58%) 3.7M	287.9s (69%) 104.0M	705.7s (70%) 211.5M
ExTrie	0.8s (50%) 2.3M	5.4s (64%) 9.7M	16.7s (70%) 23.6M
AmbExTrie	0.7s (60%) 2.1M	3.5s (68%) 10.2M	9.2s (70%) 21.8M
PairTrieStd	0.7s (63%) 1.1M	10.2s (36%) 6.0M	27.1s (42%) 14.5M
PairTrieStdCont	0.6s (59%) 0.9M	6.3s (43%) 7.1M	17.3s (47%) 16.9M
PairTrieStdStack	0.6s (61%) 1.1M	8.2s (42%) 9.5M	21.1s (51%) 19.2M
PairTrieStdEndoCont	0.6s (64%) 1.1M	6.7s (45%) 6.2M	17.7s (50%) 17.0M
PairTrieStdEndoStack	0.6s (67%) 1.1M	8.6s (50%) 10.8M	21.6s (55%) 17.5M

The trie parsers are the clear winners, tightly followed by the pairing tries. The streams are simply not very suited for ambiguous input. One interesting thing is that the endomorphisms are not good for ambiguous input. Also to note is the difference in memory usage between the backtracking parsers and the others.

The left-factorized NL parser This parser is the same grammar as before, but working on already tokenized and tagged sentences. The test data consisted of seven lines each of between 20 and 45 words. The ambiguities varied between 4 and 15000 different results for a sentence.

NLParser (LF)	Test 1	Test 2	Test 3
Standard	5.4s (16%) 1.1M	30.5s (20%) 6.1M	88.9s (16%) 10.9M
StdCont	3.6s (36%) 0.7M	22.7s (44%) 3.7M	61.5s (42%) 6.0M
StdStack	2.8s (20%) 0.9M	16.6s (22%) 4.7M	41.0s (14%) 7.3M
StdEndoCont	3.7s (37%) 0.7M	24.5s (46%) 3.6M	67.1s (43%) 6.0M
StdEndoStack	3.2s (23%) 0.9M	18.7s (28%) 4.5M	48.7s (22%) 7.7M
Stream	12.8s (44%) 4.9M	252.1s (52%) 19.3M	> 1000s
StreamCont	4.5s (54%) 6.4M	27.4s (56%) 19.4M	81.8s (60%) 109.4M
StreamStack	5.8s (64%) 8.4M	35.5s (66%) 35.8M	111.0s (70%) 183.9M
ExTrie	7.5s (43%) 3.5M	198.8s (57%) 16.1M	> 1000s
AmbExTrie	1.4s (51%) 3.6M	8.5s (54%) 16.2M	32.2s (73%) 50.5M
PairTrieStd	3.8s (19%) 1.3M	23.9s (26%) 6.8M	66.8s (20%) 10.2M
PairTrieStdCont	2.6s (18%) 1.3M	16.5s (22%) 6.4M	41.0s (14%) 10.7M
PairTrieStdStack	2.9s (20%) 1.5M	18.7s (26%) 6.3M	43.7s (16%) 11.3M
PairTrieStdEndoCont	3.3s (28%) 1.3M	21.1s (32%) 6.5M	54.2s (30%) 10.6M
PairTrieStdEndoStack	3.2s (37%) 1.4M	20.9s (37%) 7.2M	52.4s (32%) 11.5M

The winner is still the trie parsers, but only the ambiguous trie can handle this grammar. Probably the input were simple too ambiguous to the ordinary trie. Observe also that the pair trie doesn't gain anything, since the grammar is left-factorized.

The complex NL parser This is an extended version of the previous grammar, which handles conjunctions as well as some other Swedish grammatical constructs. It also works on tokenized and tagged data. The test data were taken from a real corpus and each file consisted of 300–1000 lines of between 1 and 60 words. The ambiguities varied between 1 and 100 000 different results for a sentence.

NLParser (Complex)	Test 1	Test 2	Test 3
Standard	3.8s (6%) 0.1M	48.6s (12%) 3.1M	615.5s (17%) 210.9M
StdCont	1.7s (4%) 0.0M	26.1s (12%) 1.8M	444.9s (28%) 126.8M
StdStack	2.2s (3%) 0.0M	31.2s (11%) 1.8M	485.5s (24%) 128.1M
StdEndoCont	2.0s (2%) 0.0M	29.1s (19%) 1.9M	425.6s (30%) 123.9M
StdEndoStack	2.3s (5%) 0.0M	32.7s (19%) 2.0M	453.8s (27%) 122.5M
Stream	1.7s (4%) 0.0M	34.0s (30%) 5.0M	606.6s (44%) 342.0M
StreamCont	2.0s (9%) 0.0M	42.4s (49%) 12.2M	> 1000s
StreamStack	2.2s (16%) 0.0M	47.9s (53%) 13.6M	> 1000s
ExTrie	0.8s (48%) 2.7M	12.0s (49%) 22.5M	204.6s (47%) 103.8M
AmbExTrie	0.8s (41%) 2.8M	10.2s (49%) 25.6M	90.1s (19%) 57.6M
PairTrieStd	3.6s (25%) 3.5M	84.8s (26%) 32.2M	> 1000s
PairTrieStdCont	2.5s (34%) 4.4M	56.3s (36%) 37.1M	> 1000s
PairTrieStdStack	2.6s (30%) 3.3M	67.9s (45%) 33.0M	> 1000s
PairTrieStdEndoCont	2.9s (31%) 4.5M	63.2s (32%) 34.2M	> 1000s
PairTrieStdEndoStack	2.6s (31%) 3.2M	68.2s (45%) 33.1M	> 1000s

Even in this version the ambiguous trie wins. Surprisingly the pairing tries are terribly slow on this grammar, while the backtracking parsers and the stream parsers are approximately as efficient. Once again the endomorphisms just makes things worse, which also the continuations do for the stream parser.

B.2 Parsing algorithms

We have tested the three different parsing algorithms in chapters 5, 6 and 7. The implementations are all optimized according to the guidelines in the exercises. Since the chapter on LR parsing describes three different versions – depth-first, breadth-first and Tomita parsing – we have tested all three versions.

The test data was divided into three test files for each grammar (or two for the Trivial grammar). The lengths of the sentences (which is roughly equivalent to the complexity) increased from the first file to the last file.

The Trivial grammar This grammar is a very simple grammar (only 5 productions) which is highly ambiguous. It was tested on ambiguous sentences of increasing length.

$$\begin{array}{lcl} \mathsf{S} & \longrightarrow & \mathsf{NP}\ \ \mathsf{VP} \\ \mathsf{VP} & \longrightarrow & \mathsf{Verb}\ \ \mathsf{NP}\ \ \mid\ \ \mathsf{VP}\ \ \mathsf{PP} \\ \mathsf{NP} & \longrightarrow & \mathsf{NP}\ \ \mathsf{PP} \\ \mathsf{PP} & \longrightarrow & \mathsf{Prep}\ \ \mathsf{NP} \end{array}$$

The terminal categories are NP, Verb and Prep.

Trivial	**Test 1**			**Test 2**		
Chart Parser	8.8s	(6%)	0.8M	33.0s	(7%)	2.5M
CYK Parser	8.8s	(8%)	0.9M	33.3s	(8%)	2.9M
LR Parser (T)	17.3s	(20%)	1.8M	65.0s	(22%)	6.5M
LR Parser (BF)	21.9s	(10%)	2.2M	79.4s	(10%)	7.6M
LR Parser (DF)	23.9s	(10%)	2.6M	86.0s	(10%)	8.2M

As can be noted, the LR parsers are not as efficient as the chart parser and the CYK parser. This is mainly because this simple grammar doesn't win anything on being compiled into an LR table, but also because the LR implementations are not polynomial as a real Tomita parser would be.

The Simple grammar This grammar is similar to the NL parser and consists of 19 productions. It was tested on real test data from a corpus of Swedish texts. The test data consisted of 2500 lines, each between 5 and 55 words long, and with between 0 and 130 parse results per sentence.

$$
\begin{array}{lll}
\mathsf{S} & \longrightarrow & \mathsf{S_1} \mid \mathsf{S_1\ S} \\
\mathsf{S_1} & \longrightarrow & \mathsf{NP\ VP} \\
\mathsf{VP} & \longrightarrow & \mathsf{Verb} \mid \mathsf{VP\ NP} \mid \mathsf{VP\ PP} \\
\mathsf{NP} & \longrightarrow & \mathsf{N_1} \mid \mathsf{Noun} \mid \mathsf{AP\ Noun} \\
 & \mid & \mathsf{Det\ Noun} \mid \mathsf{Det\ AP\ Noun} \\
 & \mid & \mathsf{NP\ PP} \mid \mathsf{NP\ RP} \\
\mathsf{N_1} & \longrightarrow & \mathsf{Name} \mid \mathsf{Pron} \\
\mathsf{AP} & \longrightarrow & \mathsf{Adj} \mid \mathsf{Adj\ AP} \\
\mathsf{PP} & \longrightarrow & \mathsf{Prep\ NP} \\
\mathsf{RP} & \longrightarrow & \mathsf{Relpro\ VP}
\end{array}
$$

The terminal categories are Adj, Det, Name, Noun, Prep, Pron, Relpro and Verb.

Simple	**Test 1**			**Test 2**			**Test 3**		
Chart Parser	3.1s	(6%)	0.5M	4.3s	(12%)	0.7M	1.3s	(8%)	0.2M
LR Parser (T)	1.2s	(3%)	0.4M	1.6s	(8%)	0.6M	0.4s	(8%)	0.0M
LR Parser (BF)	1.2s	(10%)	0.4M	1.9s	(11%)	0.8M	0.7s	(2%)	0.0M
LR Parser (DF)	1.3s	(8%)	0.4M	2.2s	(7%)	0.8M	0.8s	(8%)	0.0M

The Simple grammar in CNF The grammar was also translated to Chomsky Normal Form, to be able to measure the performance of the CYK algorithm. The number of productions decreased from 19 to 14, but instead more categories became terminal.

Simple (CNF)	**Test 1**			**Test 2**			**Test 3**		
Chart Parser	3.3s	(7%)	0.5M	4.6s	(9%)	0.8M	1.4s	(7%)	0.2M
CYK Parser	2.2s	(6%)	0.6M	3.6s	(5%)	0.7M	1.3s	(5%)	0.0M
LR Parser (T)	1.1s	(7%)	0.4M	1.5s	(7%)	0.6M	0.4s	(4%)	0.0M
LR Parser (BF)	1.1s	(10%)	0.4M	1.8s	(9%)	0.8M	0.6s	(6%)	0.0M
LR Parser (DF)	1.3s	(7%)	0.4M	2.2s	(6%)	0.8M	0.8s	(4%)	0.0M

The LR parsers are quite a lot faster than the other algorithms, which is a bit surprising, since the implementations are exponential in the length of the input. But this grammar is apparently very well suited for compilation into an LR table. The CYK parser is faster than the chart parser, which probably is because of its very simple and straightforward implementation that doesn't introduce any overhead.

The Complex grammar The Complex grammar is an extended version of the previous grammar, with 31 productions. This grammar recognizes more sentences, but is also much more ambiguous. The grammar was tested on the same corpus as before (but with more complex sentences). The sentences were between 0 and 60 words in length, and had between 0 and 30 000 parse results per sentence.

$$\begin{array}{lll}
\mathsf{S}_1 & \longrightarrow & \ldots \mid \mathsf{NP} \mid \mathsf{VP} \mid \mathsf{S}_1\ \mathsf{Conj}\ \mathsf{S}_1 \\
\mathsf{VP} & \longrightarrow & \ldots \mid \mathsf{VP}\ \mathsf{Subj}\ \mathsf{S}_1 \mid \mathsf{VP}\ \mathsf{IP} \mid \mathsf{VP}\ \mathsf{Conj}\ \mathsf{VP} \\
\mathsf{Verb} & \longrightarrow & \mathsf{Verb}\ \mathsf{Part} \\
\mathsf{NP} & \longrightarrow & \ldots \mid \mathsf{NP}\ \mathsf{Conj}\ \mathsf{NP} \\
\mathsf{N}_1 & \longrightarrow & \ldots \mid \mathsf{Name}\ \mathsf{N}_1 \mid \mathsf{Pron}\ \mathsf{N}_1 \mid \mathsf{Number}\ \mathsf{N}_1 \\
\mathsf{IP} & \longrightarrow & \mathsf{Inf}\ \mathsf{VP} \\
\mathsf{AP} & \longrightarrow & \ldots \\
\mathsf{PP} & \longrightarrow & \ldots \\
\mathsf{RP} & \longrightarrow & \ldots
\end{array}$$

The grammar also consists of the productions from the Simple grammar. The new terminal categories are Conj, Inf, Part and Subj.

Complex	**Test 1**			**Test 2**			**Test 3**		
Chart Parser	4.1s	(7%)	0.2M	19.4s	(8%)	0.9M	187.5s	(3%)	21.7M
LR Parser (T)	2.9s	(2%)	0.3M	26.6s	(8%)	1.4M		> 1000s	
LR Parser (BF)	4.0s	(3%)	0.3M	60.6s	(6%)	2.2M		> 1000s	
LR Parser (DF)	4.5s	(4%)	0.3M	68.4s	(5%)	3.4M		> 1000s	

The Complex grammar in CNF The Chomsky Normal Form version of the Complex grammar has 63 productions, which means that the timings are different from the original version.

Complex (CNF)	**Test 1**			**Test 2**			**Test 3**		
Chart Parser	6.8s	(6%)	0.3M	25.5s	(12%)	0.9M	147.9s	(5%)	26.8M
CYK Parser	2.5s	(2%)	0.0M	15.5s	(5%)	0.9M	141.4s	(4%)	18.3M
LR Parser (T)	3.5s	(4%)	0.3M	24.4s	(11%)	1.4M		> 1000s	
LR Parser (BF)	4.9s	(3%)	0.2M	66.1s	(4%)	1.9M		> 1000s	
LR Parser (DF)	6.3s	(3%)	0.3M	88.6s	(4%)	2.7M		> 1000s	

Once again the CYK parser outperforms the chart parser, at least on the simple input. We can see that the approximation of the Tomita parser is faster than the chart parser on simple input – i.e. input without much ambiguities. But as soon as the sentences become more ambiguous, the LR algorithms slows down. This is of course because of their exponential complexity.

Bibliography

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers – Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] William H Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.

[3] Magnus Carlsson and Thomas Hallgren. *Fudgets – Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1998.

[4] Manuel Chakravarty. Lazy lexing is fast. In A Middeldorp and T Sato, editors, *Fourth Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *LNCS*, pages 68–84. Springer Verlag, 1999.

[5] Koen Claessen. Parsek – fast and space-efficient monadic parser combinators, 2001. Available from the Chalmers Multi Library:
`http://www.cs.chalmers.se/Cs/Research/Functional/MultiLib`

[6] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *ASIAN Computer Science Proceedings*, Phuket, Thailand, 1999. ACM SIGPlan.

[7] Byron Cook and John Launchbury. Disposable memo functions. In *ACM SIGPLAN Haskell Workshop*, June 1997.

[8] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[9] *The GHC User's Guide*, 2001. Available from the Haskell web site:
`http://www.haskell.org/ghc`

[10] Andy Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, Denmark, 1993.

[11] Susan Graham, Michael Harrison, and Walter Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, 1980.

[12] Jörgen Gustavsson. *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages.* PhD thesis, Göteborg University, 2001.

[13] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[14] Paul Hudak. Modular domain specific languages and tools. In *Fifth International Conference on Software Reuse*, pages 134–142, Victoria, Canada, June 1998.

[15] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98. Technical report, Yale University, October 1999. Available from the Haskell web site:
`http://www.haskell.org/tutorial`

[16] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323– 343, July 1992.

[17] Graham Hutton and Erik Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[18] Mark Jones. Type classes with functional dependencies. In *9th European Symposium on Programming*, Berlin, Germany, 2000. Springer-Verlag LNCS 1782.

[19] Tadao Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical report, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[20] Martin Kay. Algorithm schemata and data structures in syntactic processing. In Barbara Grosz, Karen Jones, and Bonnie Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufman Publishers, Los Altos, CA, 1986.

[21] James Kilbury. Chart parsing and the Earley algorithm. In Ursula Klenk, editor, *Kontextfreie Syntaxen und wervandte Systeme.* Niemeyer, Tübingen, Germany, 1985.

[22] Donald Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[23] Pieter Koopman and Rinus Plasmeijer. Efficient combinator parsers. In A J T Davie K Hammond and C Clack, editors, *Implementation of Functional Languages*, volume 1595 of *LNCS*, pages 122–138. Springer Verlag, 1998.

[24] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In J Loeckx, editor, *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 255–269. Springer-Verlag, 1974.

[25] Konstantin Läufer. *Polymorphic Type Inference and Abstract Data Types.* PhD thesis, New York University, 1992.

[26] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht, Netherlands, 2001.

[27] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95*, San Fransisco, January 1995. ACM.

[28] Simon Marlow and Andy Gill. *Happy User Guide*, 2001. Available from the Haskell web site:
`http://www.haskell.org/happy`

[29] Ben Medlock. A tool for generalised LR parsing in Haskell. Master's thesis, Department of Computer Science, University of Durham, UK, To appear 2002. Contact Paul Callaghan, `p.c.callaghan@durham.ac.uk`

[30] Mark-Jan Nederhof and Janos Sarbo. Increasing the applicability of LR parsing. In Harry Bunt and Masaru Tomita, editors, *Recent Advances in Parsing Technology*, pages 35–58. Kluwer Academic Publishers, 1996.

[31] Rohman Nozohoor-Farshi. Handling of ill-designed grammars in Tomita's parsing algorithm. In *International Workshop on Parsing Technologies*, pages 182–192, Pittsburgh, PA, 1989. Carnegie Mellon University.

[32] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[33] Nigel Perry. *The Implementation of Practical Functional Programming Languages.* PhD thesis, Imperial College, London, 1990.

[34] Simon Peyton Jones. Tackling the awkward squad. Technical report, Microsoft Research, Cambridge, February 2001. Presented at the 2000 Marktoberdorf Summer School.

[35] Simon Peyton Jones (editor). Report on the programming language Haskell 98, a non-strict purely functional language. Technical report, Yale University, February 1999. Available from the Haskell web site:
`http://www.haskell.org/definition`

[36] Niklas Röjemo. *Garbage Collection and Memory Efficiency in Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1995.

[37] Stuart Shieber, Yves Schabes, and Fernando Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.

[38] Klaas Sikkel. *Parsing Schemata.* Springer Verlag, 1997.

[39] Doaitse Swierstra. Combinator parsers: From toys to tools. In Graham Hutton, editor, *Haskell Workshop*, pages 46–57, September 2000.

[40] Doaitse Swierstra and Pablo Azero Alcocer. Fast, error correcting parser combinators. In J Pavella, G Tel, and M Bartosek, editors, *SOFSEM99 Theory and Practice of Informatics*, volume 1725 of *LNCS*, pages 111–129. Springer-Verlag, November 1999.

[41] Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 1–17. Springer-Verlag, 1996.

[42] Masaru Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Press, 1986.

[43] Leslie Valiant. General context-free recognition in less than cubic time. *Journal of Computer and Systems Sciences*, 10(2):308–315, April 1975.

[44] Philip Wadler. How to replace failure by a list of successes. In *Second Int. Conf. on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Springer-Verlag.

[45] Philip Wadler. Monads for functional programming. In *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and Systems Sciences.* Springer-Verlag, August 1992.

[46] Mats Wirén. *Studies in Incremental Natural-Language Analysis.* PhD thesis, Linköping University, Linköping, Sweden, 1992.

[47] Daniel H Younger. Recognition of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.